TASKING VX-toolset for Power Architecture User Guide

MA170-053 (v2.0) March 29, 2012

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 VX-toolset for Power Architecture™ C compiler fully supports the ISO-C standard and adds extra possibilities to program the special functions of the target.

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

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

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

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

1.1. Data Types

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

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	$[0, 2^{32}-1]$

C Type	Size	Align	Limits
enum [*]	32 16 8	32 16 8	$[-2^{31}, 2^{31}-1]$ $[-2^{15}, 2^{15}-1]$ or $[0, 2^{16}-1]$ $[-2^{7}, 2^{7}-1]$ or $[0, 2^{8}-1]$
long	32	32	$[-2^{31}, 2^{31}-1]$
unsigned long	32	32	$[0, 2^{32}-1]$
long long	64	64	$[-2^{63}, 2^{63}-1]$
unsigned long long	64	64	$[0, 2^{64}-1]$
float (23-bit mantissa)	32	32	[–3.402E+38, –1.175E-38] [+1.175E-38, +3.402E+38]
double long double (52-bit mantissa)	64	64	[-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	64	[-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	64	real part + imaginary part
pointer to data or function	32	32	$[0, 2^{32}-1]$

[*] When you use the `enum` type, the compiler by default uses 32-bit integers for enumeration. If you use C compiler option **--small-enumeration** the compiler will use the smallest sufficient integer type (`char`, `short` or `int`).

1.2. Accessing Memory

There are three ways to allocate a variable in a particular memory space. They are listed below, the method with the highest priority is listed first.

1. Memory qualifiers.
2. Allocation options or pragmas.
3. Threshold options.

When one of the above methods does not apply to an object, it will be allocated in the `__data` memory space.

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

1.2.1. Memory Qualifiers

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

The Power Architecture has small data memories, sdata, sdata2 and sdata0, all with a size of 64kB. By default, all global and static data objects smaller than 4 bytes are placed in sdata or sdata2 (non-constant data is placed in sdata whereas constant data is placed in sdata2). With memory qualifiers you can overrule this default.

Memory qualifiers cannot be used to qualify a pointer.

You can specify the following memory qualifiers:

Qualifier	Description	Memory type	Maximum object size	Location	Base register	Section type
__data	Normal initialized data	ROM** / RAM	Size of available memory	Anywhere in memory	--	.data
__data	Normal non-initialized data	RAM	Size of available memory	Anywhere in memory	--	.bss
__sdata	Short addressable initialized data	RAM	64 kB *	Anywhere in memory but concatenated to .sbss	r13	.sdata
__sdata	Short addressable non-initialized data	RAM	64 kB *	Anywhere in memory but concatenated to .sdata	r13	.sbss
__sdata2	Short addressable initialized data	ROM** / RAM	64 kB *	Anywhere in memory but concatenated to .sbss2	r2	.sdata2
__sdata2	Short addressable non-initialized data	RAM	64 kB *	Anywhere in memory but concatenated to .sdata2	r2	.sbss2
__sdata0	Short addressable initialized data in zero-page	ROM*** / RAM	64 kB *	+/- 32 kB offset from address 0	r0	.sdata0
__sdata0	Short addressable non-initialized data in zero-page	RAM	64 kB *	+/- 32 kB offset from address 0	r0	.sbss0

* The maximum size of all objects in .sdata/.sbss, .sdata2/.sbbs2 and .sdata0/.sbss0 cannot exceed 64 kB.

** Constants are allocated in a romdata section.

*** With compiler option **--eabi=+sdata0-rom** constants in `__sdata0` are allocated in a romdata section.

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

Examples using memory qualifiers

To declare a fast accessible integer in directly addressable memory:

```
long long l = 1234;                // long long reserved in .data (by default)
const long long k = 1234;          // long long reserved in .data,
                                   // romdata (by default)

__sdata  long long  m;             // long long reserved in .sdata
const __sdata2 long long n = 1234;  // long long in .sdata2, romdata
```

The memory type qualifiers are treated like any other data type specifier (such as unsigned). This means the examples above can also be declared as:

```
long long  __sdata  m = 1234;
const long long __sdata2  n = 1234;
```

You cannot use memory qualifiers in structure declarations:

```
struct S {
   __data  int   i;  /* put an integer in data
                        memory: Incorrect !        */
   __sdata int * p;  /* put an integer pointer in
                        sdata memory: Incorrect !   */
};
```

1.2.2. Data Allocation Options and Pragmas

When an object does not have an explicit memory qualifier, you can use one of the following options or pragmas to assign a memory space.

- C compiler option **--data-memory** / pragma `data_memory`
- C compiler option **--constant-data-memory** / pragma `constant_data_memory`
- C compiler option **--string-literal-memory** / pragma `string_literal_memory`

Note that you cannot explicitly qualify string literals, you can only place string literals in a particular memory space with C compiler option **--string-literal-memory** or with pragma `string_literal_memory`.

1.2.3. Data Allocation Threshold Options

When you have not used memory qualifiers and/or allocation options or pragmas for data allocation, the compiler tries to place an object in a particular space automatically by testing the object's size and type against threshold settings. The possible threshold settings are:

Threshold	Object type selection	Assigned qualifier
--default-sdata-size=[*min*,]*max*	**--default-sdata-type=***type*	`__sdata`
--default-sdata2-size=[*min*,]*max*	**--default-sdata2-type=***type*	`__sdata2`
--default-sdata0-size=[*min*,]*max*	**--default-sdata0-type=***type*	`__sdata0`

When an object's size is greater than *min* and less than or equal to *max*, and the object's type matches the selected type(s) of a threshold setting, the indicated qualifier will be assigned. When multiple threshold settings apply to the same object, a warning will be issued. You can select constant or non-constant objects as object type. Instead of the options you can also use the pragma versions of the options to select the object types.

Examples of allowed declarations

```
/*
 *  Used compiler options:
 *
 *  a) --default-sdata-size=0,8    --default-sdata-type=Cn
 *  b) --default-sdata2-size=0,8   --default-sdata2-type=cN
 *  c) --default-sdata0-size=0,0   --default-sdata0-type=cN
 *  d) --data-memory=threshold
 */
float f;
#pragma data_memory __sdata
__data float  f;             /* explicit qualifier overrules pragma
                                and previous threshold allocation;
                                allocated in __data memory space    */
double        d;             /* no explicit memory qualifier
                                pragma assigns memory qualifier;
                                allocated in __sdata memory space   */
long          l;             /* no explicit memory qualifier
                                pragma assigns memory qualifier;
                                allocated in __sdata memory space   */
#pragma default_sdata_type CN  /* do not allow non-constant objects to
                                be allocated in __sdata2 space automatically */
#pragma data_memory threshold
long          l;             /* threshold settings not relevant because
                                already explicitly qualified in previous
                                declaration;
                                allocated in __sdata memory space   */
#pragma default_sdata_type Cn
```

```
float         f;              /* no explicit memory qualifier,
                                 inherit qualifier from previous declaration;
                                 allocated in __data memory space    */
double        d;              /* no explicit memory qualifier, inherit qualifier
                                 assigned by pragma from previous declaration;
                                 allocated in __sdata memory space   */
int           a[25];          /* no explicit memory qualifier,
                                 object size outside any threshold range;
                                 allocated in __data memory space    */
```

Examples of redeclaration errors

```
/*
 *  Used compiler options:
 *
 *  a) --default-sdata-size=0,8    --default-sdata-type=Cn
 *  b) --default-sdata2-size=0,8   --default-sdata2-type=cN
 *  c) --default-sdata0-size=0,0   --default-sdata0-type=cN
 *  d) --data-memory=threshold
 */
__data float    f;
__sdata double  d;
__sdata2 double d;            /* error; this declaration is in __sdata2,
                                 while the previous declaration was in __sdata */
#pragma data_memory __sdata
float           f;            /* error; this declaration is in __sdata,
                                 while the previous declaration was in __data  */
__data int      a;
int             a;            /* error: previous declaration is in __data,
                                 while the data_memory pragma assigns __sdata
                                 to this declaration  */
#pragma data_memory threshold
#pragma default_sdata_type CN
int             i;            /* allocated in __data because no threshold
                                 setting applies to non-constant objects       */
#pragma default_sdata_type Cn
int             i;            /* error; this declaration is allocated in
                                 __sdata because of the threshold settings,
                                 while the previous declaration was in __data  */
```

1.2.4. Allocation of Uninitialized Constants

Uninitialized constant data is treated like it is initialized with 0 (zero).

For example:

```
const int i;
```

is treated as:

```
const int i = 0;
```

Depending upon the **--default-sdata*-size** options these variables will be allocated in a `.rodata` or `.sdata[02], romdata` section.

With the default compiler settings, the example above will be allocated as:

```
    .section .sdata2.t.i, romdata
    .global  i
    .align   4
    .size    i, 4
i:  .type    object
    .db    0
    .db    0
    .db    0
    .db    0
    .endsec
```

When `#pragma noclear` is used on an uninitialized constant object, it will be moved to an `.[s]bss[02]` section. This section will have the `noclear` attribute. However, it is unwanted to have `.sbss0` or `.sbbs2` sections for the following reasons:

- In the `__sdata0` memory space not all targets have RAM memory available.
- For the `__sdata2` memory space the EABI prescribes that when ROM and RAM sections are mixed, ROM sections must be converted to initialized RAM sections. This is unwanted, because this uses twice the amount of memory. Once for the ROM copy, and once for the initialized RAM section. It also costs time because the startup code has to copy the ROM section to the RAM section at startup.

For these reasons `#pragma noclear` is ignored for objects that are allocated in the `__sdata0` or `__sdata2` space automatically (i.e. by the threshold settings), and the compiler generates a warning. To prevent this warning use one of the following ways:

- Use an explicit qualifier to locate the object in the `__data` or `__sdata` space. Or, if you wish to ignore the issues mentioned above you can even use `__sdata0` or `__sdata2` explicitly. It is also possible to use the C compiler option **--constant-data-memory** or `#pragma constant_data_memory` to assign an explicit qualifier.
- Change the threshold settings such that the object will be allocated in the `__data` or `__sdata` space automatically.

Note that these restrictions do not apply to the `__sdata0` space when the C compiler option **--eabi=-sdata0-rom** is used.

1.2.5. 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.

Examples

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

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

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

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

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

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

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

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

General syntax of the __asm keyword

```
__asm( "instruction_template"
       [ : output_param_list
```

```
        [ : input_param_list
        [ : register_save_list]]] );
```

instruction_template	Assembly instructions that may contain parameters from the input list or output list in the form: **%***parm_nr*
%*parm_nr*	Parameter number in the range 0 .. 9.
output_param_list	[[**"=**[**&**]*constraint_char***"(***C_expression***)**],...]
input_param_list	[[**"***constraint_char***"(***C_expression***)**],...]
&	Says that an output operand is written to before the inputs are read, so this output must not be the same register as any input.
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_save_list	[["*register_name*"],...]
register_name	Name of the register you want to reserve. Note that saving too many registers can make register allocation impossible.

Specifying registers for C variables

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

You can reserve the registers that are used in the assembly instructions, either in the parameter lists or in the reserved register list (*register_save_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
r	general purpose register	r0, r3 - r12, r14 - r31	
v	VLE general purpose register	r0, r3 - r7, r24 - r31	
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. 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. 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 the variable. The compiler generates code to assign the result to the output variable.

```
char out;

void func(void)
{
    __asm( "li  %0,0xff" : "=r"(out));
}
```

Generated assembly code:

```
    li  r11,0xff
    stb r11,@relsda(out)(r13)
```

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( "li   r0,%0"
           :
           : "r" (in) );
}
```

Generated assembly code:

```
    lwz   r0,@relsda(in)(r13)
    li    r0,r0
```

Example 4: using input and output parameters

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

Generated assembly code:

```
    lwz     r0,@relsda(in1)(r13)
    lwz     r11,@relsda(in2)(r13)
    add     r11, r0, r11
    st.w    r11,@relsda(out)(r13)
```

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 (register `r12` is used instead).

```
int in1, in2, out;

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

Generated assembly code:

```
    lwz     r11,@relsda(in1)(r13)
    lwz     r12,@relsda(in2)(r13)
    add    r11, r11, r12
    st.w   r11,@relsda(out)(r13)
```

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(
    "slwi   %1,%1,2\n\t"
    "add    %2,%2,%1\n\t"
    "stw    %2,0(%5)\n\t"
    "mr     %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:

```
  li    r0,1000
  mr    r11,r0

  la    r10,@relsda(parm)(r13)
  slwi  r0,r0,2
  add   r11,r11,r0
  stw   r11,0(r10)
  mr    r12,r11

  stw   r12,@relsda(result)(r13)
```

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(
    "slwi   %1,%1,2\n\t"
    "add    %2,%2,%1\n\t"
    "stw    %2,0(%3)\n\t"
    "mr     %0,%2"
    : "=&r" (retvalue)
    : "r" (par1), "r" (par2), "r" (par3)
);
```

Generated assembly code:

```
  la    r11,@relsda(parm)(r13)
  li    r12,1000
  slwi  r12,r12,2
  add   r12,r12,r12  ; same register, but is expected read-only
  stw   r12,0(r11)
  mr    r0,r12

  stw   r0,@relsda(result)(r13)   ; contains unexpected result
```

1.4. 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.

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`'.

base

Marks a non-critical interrupt class. For more information see Section 1.8.6, *Interrupt Functions*.

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.

critical

Marks a critical interrupt class. For more information see Section 1.8.6, *Interrupt Functions*.

debug

Marks a debug interrupt class. For more information see Section 1.8.6, *Interrupt Functions*.

enable

Specifies that the compiler also generates code to re-enable interrupts of the same class. For more information see Section 1.8.6, *Interrupt Functions*.

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 */
```

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.

interrupt([*vector*])

You can use `__attribute__((interrupt(vector)))` to indicate that the specified function is an interrupt handler. The C compiler generates function entry and exit sequences suitable for use in an interrupt handler when this attribute is present.

For more information see Section 1.8.6, *Interrupt Functions*.

machine

Marks a machine check interrupt class. For more information see Section 1.8.6, *Interrupt Functions*.

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.

nested

Specifies a nested interrupt function. For more information see Section 1.8.6, *Interrupt Functions*.

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.

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`.

push_cr

Push the condition register on the stack. See also the `__get_cr()` intrinsic function. Can only be used with an interrupt function.

push_gpr32

Push all 32-bit registers in one consecutive area on the stack. See also the `__get_gpr32()` intrinsic function. Can only be used with an interrupt function.

push_srr

Push the save-restore registers on the stack. See also the `__get_srr()` intrinsic function. Can only be used with an interrupt function.

reset

Marks a reset interrupt class. For more information see Section 1.8.6, *Interrupt Functions*.

section("*section_name*")

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

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

puts the function `foobar` in the section named `bar`.

See also `#pragma section`.

used

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

```
static const char copyright[] __attribute__((used)) = "Copyright 2012 Altium 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`.

1.5. Pragmas to Control the Compiler

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

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

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;

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

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

Supported pragmas

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

alias *symbol=defined_symbol*

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

clear [on | off | default | restore] / noclear

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

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

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

This pragma sets the `init` section attribute for `.text` and `.text_vle` sections. As a result, the startup code will copy code from ROM to RAM, from where the code will be executed.

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

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

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

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

With this pragma `const` variables are allocated in an initialized data section.

constant_data_memory {*space* | default | restore} (*)

Controls the allocation of constant data objects. Same as C compiler option **--constant-data-memory**.

data_memory {*space* | default | restore} (*)

Controls the allocation of non-constant data objects. Same as C compiler option **--data-memory**.

default_sdata_type {*flag*,.. | default | restore} (*)

Selects the type of objects that are affected by C compiler option **--default-sdata-size**. Same as C compiler option **--default-sdata-type**.

default_sdata0_type {*flag*,.. | default | restore} (*)

Selects the type of objects that are affected by C compiler option **--default-sdata0-size**. Same as C compiler option **--default-sdata0-type**.

default_sdata2_type {*flag*,.. | default | restore} (*)

Selects the type of objects that are affected by C compiler option **--default-sdata2-size**. Same as C compiler option **--default-sdata2-type**.

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.

inline / noinline / smartinline

See Section 1.8.3, *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 enable 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.

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 3.8.2, *C Code Checking: MISRA-C*.

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

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

See Section 3.6, *Compiler Optimizations*.

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

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

runtime [*flags* | default | restore]

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

section [[*type*=|*whitespace*][*format_string*],... | default | restore] / endsection

Rename sections by adding a *format_string* to all section names specified with *.type*, or restore default section naming. If you specify only a *format_string* (without a type), the suffix is added to all section names.

See Section 1.9, *Section Naming*, C compiler option **--rename-sections** and assembler directive `.SECTION` for more information.

small_enumeration [on | off | default | restore]

This pragma controls the type of enum types. Same as C compiler option **--small-enumeration**

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

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

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

stdinc [on | off | default | restore]

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

string_literal_memory {*space* | default | restore} (*)

Controls the allocation of string literals. Same as C compiler option **--string-literal-memory**.

switch auto | jump_tab | linear | lookup | restore

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

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

tradeoff *level* [default | restore]

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

unroll_factor *value* | default | restore] / endunroll_factor

Specify how many times the following loop should be unrolled, if possible. At the end of the loop use `endunroll_factor`.

See C compiler option **--unroll-factor**.

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

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

weak *symbol*

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

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

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

1.6. Predefined Preprocessor Macros

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

Macro	Description
__BIG_ENDIAN__	Expands to 1 if big-endian mode is selected (option **--endianness=big**), otherwise unrecognized as macro. This is the default.
__BUILD__	Identifies the build number of the compiler, composed of decimal digits for the build number, three digits for the major branch number and three digits for the minor branch number. For example, if you use build 1.22.1 of the compiler, __BUILD__ expands to 1022001. If there is no branch number, the branch digits expand to zero. For example, build 127 results in 127000000.
__CPPC__	Identifies the compiler. You can use this symbol to flag parts of the source which must be recognized by the TASKING **cppc** compiler only. It expands to 1.
__CORE__	Expands to the name of the core depending on the C compiler option **--core**. The symbol expands to `e200z0` when no **--core** is supplied.
__CORE_*core*__	A symbol is defined depending on the option **--core**. The *core* is converted to upper case. For example, if **--core=e200z4** is specified, the symbol `__CORE_E200Z4__` is defined. When no **--core** is supplied, the compiler defines `__CORE_E200Z0__`.
__DATE__	Expands to the compilation date: “mmm dd yyyy”.
__DOUBLE_FP__	Expands to 1 if you did not use option **--no-double** (Treat ‘double’ as ‘float’), otherwise unrecognized as macro.
__FILE__	Expands to the current source file name.
__FPU__	Expands to 1 when the selected core has a floating-point unit and the option **--no-fpu** is not used. Otherwise unrecognized as a macro.
__LINE__	Expands to the line number of the line where this macro is called.
__LITTLE_ENDIAN__	Expands to 1 if little-endian mode is selected (option **--endianness=little**), otherwise unrecognized as macro.
__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
__SINGLE_FP__	Expands to 1 if you used option **--no-double** (Treat ‘double’ as ‘float’), otherwise unrecognized as macro.

Macro	Description
__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_VERSION__	Identifies the ISO-C version number. Expands to 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"
__VERSION__	Identifies the version number of the 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).
__VLE__	Expands to 1 when the selected core supports VLE encoded instructions and the C compiler option **--no-vle** is not used. Otherwise unrecognized as a macro.
__VX__	Identifies the VX-toolset C compiler. Expands to 1.

Example

```
#ifdef __VLE__
/* this part is only valid if VLE instructions are supported */
...
#endif
```

1.7. Switch Statement

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

A *jump chain* is comparable with an if/else-if/else-if/else construction. A *jump table* is table filled with target addresses for each possible switch value. The switch argument is used as an index within this table. A jump table can be generated in two forms: either the table contains absolute addresses or the table contains 16-bit offsets relative to a reference label in the code. A *lookup table* is a table filled with a value to compare the switch argument with and a target address to jump to. A binary search lookup is performed to select the correct target address.

By default, the compiler will automatically choose the most efficient switch implementation based on code and data size and execution speed. With the C compiler option **--tradeoff** you can tell the compiler to put more emphasis on speed than on ROM size.

Especially for large switch statements, the jump table approach executes faster than the lookup table approach. Also the jump table has a predictable behavior in execution speed: independent of the switch argument, every case is reached in the same execution time. However, when the case labels are distributed far apart, the jump table becomes sparse, wasting code memory. The compiler will not use the jump table method when the waste becomes excessive.

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

Example

The following example is used to explain the different switch statement implementations:

```
int   g;
void  f( int c )
{
    switch ( c )
    {
        case 1:
            g = 3;
            break;
        case 2:
            g = 5;
            break;
        case 10:
            g = 2;
            break;
        default:
            g = -1;
            break;
    }
    return;
}
```

Jump chain

With C compiler option **--switch=linear** the example is generated as:

```
 f:      .type   func
        cmpi    cr0,0,r3,1   ; comparable if/else construction
        beq     cr0,.L3
        cmpi    cr0,0,r3,2
        beq     cr0,.L4
        cmpi    cr0,0,r3,10
        beq     cr0,.L5
        bmaski  r0,0
        b       .L10
.L3:
        li      r0,3
        b       .L6
.L4:
        li      r0,5
        b       .L7
.L5:
        li      r0,2
.L10:
.L6:
```

```
.L7:
        stw     r0,@relsda(g)(r13)
        blr
```

Lookup table

With C compiler option **--switch=lookup** the example is generated as:

```
 f:      .type   func
        stwu    r1,-8(r1)
        mflr    r0
        stw     r0,12(r1)
        lis     r4,@ha(.1.swt+8)
        la      r4,@lo(.1.swt+8)(r4)
        bl      __bswitch32
        mtlr    r4
        blr
.L3:
        li      r0,3
        b       .L6
.L4:
        li      r0,5
        b       .L7
.L5:
        li      r0,2
        b       .L8
.L2:
        bmaski  r0,0
.L6:
.L7:
.L8:
        stw     r0,@relsda(g)(r13)
        lwz     r0,12(r1)
        mtlr    r0
        lwz     r1,0(r1)
        blr

        .section        .data.switch.$1$swt, romdata
        .align  4
        .size   .1.swt, 32
.1.swt: .type   object       ; lookup table holds switch values
        .dw     3,.L2        ; and target address to jump to
        .dw     1,.L3
        .dw     2,.L4
        .dw     10,.L5
        .endsec
```

Jump table: absolute

When a switch is implemented using a jump table, the switch argument is used as an index into a table which holds the target address. With an absolute jump table the jump table holds absolute addresses. This is the fastest jump table implementation, but it also uses more memory than the variant using 16-bit offset relative to a reference label. Therefore, this variant will be used when speed is more important than size.

With C compiler option **--switch=jump -t1** the example is generated as:

```
f:      .type   func
        addi    r4,r3,-1
        cmpli   cr0,0,r4,10
        bge     cr0,.L2
        slwi    r4,r4,2
        addis   r4,r4,@ha(.1.swt)
        lwz     r4,@lo(.1.swt)(r4)
        mtctr   r4
        bctr
.L3:
        li      r0,3
        b       .L6
.L4:
        li      r0,5
        b       .L7
.L5:
        li      r0,2
        b       .L8
.L2:
        bmaski  r0,0
.L6:
.L7:
.L8:
        stw     r0,@relsda(g)(r13)
        blr

        .section        .data.switch.$1$swt, romdata
        .align  4
        .size   .1.swt, 40
.1.swt: .type   object      ; jump table with absolute addresses
        .dw     .L3
        .dw     .L4
        .dw     .L2
        .dw     .L2
        .dw     .L2
        .dw     .L2
        .dw     .L2
        .dw     .L2
        .dw     .L2
        .dw     .L5
        .endsec
```

Jump table: relative

With a relative jump table the jump table holds 16-bit offsets relative to a reference label. This variant is slightly slower compared to an absolute jump table, but uses less memory, because the jump table is only half the size of an absolute table. Therefore, this variant will be used when the focus is more towards size.

With C compiler option **--switch=jump -t2** the example is generated as:

```
f:      .type   func
        addi    r4,r3,-1
        cmpli   cr0,0,r4,10
        bge     cr0,.L2
        slwi    r4,r4,1
        addis   r4,r4,@ha(.2.swt)
        lha     r4,@lo(.2.swt)(r4)
        addis   r4,r4,@ha(.L10)
        la      r4,@lo(.L10)(r4)
        mtctr   r4
        bctr
.L10:
.L3:
        li      r0,3
        b       .L6
.L4:
        li      r0,5
        b       .L7
.L5:
        li      r0,2
        b       .L8
.L2:
        bmaski  r0,0
.L6:
.L7:
.L8:
        stw     r0,@relsda(g)(r13)
        blr

        .section        .data.switch.$2$swt, romdata
        .align  2
        .size   .2.swt, 20
.2.swt: .type   object      ; jump table with relative addresses
        .dh     .L3-.L10
        .dh     .L4-.L10
        .dh     .L2-.L10
        .dh     .L2-.L10
        .dh     .L2-.L10
        .dh     .L2-.L10
        .dh     .L2-.L10
        .dh     .L2-.L10
        .dh     .L2-.L10
        .dh     .L5-.L10
        .endsec
```

Automatic mode

With **--switch=auto** (this is the default), the compiler automatically selects the best method to implement a switch statement. The method is chosen based upon the properties of a switch statement and the selected setting for the speed/size tradeoff. Properties of a switch statement are the number of case labels, and the density. The density is calculated as:

```
  size = max - min + 1

             cases * 100
  density = -------------- [%]
               size
```

```
Where,

max   : the maximum value of all case labels.
min   : the minimum value of all case labels.
cases : the number of case labels.
```

When the tradeoff is set to **-t4**, the compiler selects the method that uses as little memory as possible. To determine the amount of memory, the code size as well as the table size is taken into account.

For the tradeoff values **-t3** - **-t0** the focus moves more towards speed. In general this means that jump tables will be used, but not at all cost. When the density of a jump table is too low it will generally not be used. There are however cases where a jump table is both the smallest and the fastest. In that case the compiler will use that variant. The density threshold depends upon the selected tradeoff level:

Trade-off value	Density absolute jump table	Density relative jump table
0	>=20	>=10
1	>=40	>=20
2	>=60	>=30
3	>=80	>=40

When the density of a switch statement is below the threshold the compiler will most likely use a jump chain, or a lookup table. In general a lookup table only becomes useful when a switch has a large number of cases (>= 70) and the density is below 25%.

How to overrule the default switch method

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

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

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

Example:

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

#pragma switch jumptab

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

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

1.8. Functions

1.8.1. Calling Convention

Parameter passing

A lot of execution time of an application is spent transferring parameters between functions. The fastest parameter transport is via registers. Therefore, function parameters are first passed via registers. If no more registers are available for a parameter, the compiler pushes parameters on the stack.

Parameter	Register
Arithmetic 8-bit	r3-r10
Arithmetic 16-bit	r3-r10
Arithmetic 32-bit	r3-r10
Arithmetic 64-bit	r3r4, r5r6, r7r8, r9r10
Pointer	r3-r10
_Complex double	r3r4r5r6, r4r5r6r7, r5r6r7r8, r6r7r8r9, r7r8r9r10

Registers available for parameter passing are r3-r10. Up to 8 arithmetic types and/or 8 pointers can be passed this way. A 64-bit argument is passed in an even/odd data register pair. Parameter registers skipped because of alignment for a 64-bit argument are not used by subsequent 32-bit arguments. Any remaining function arguments are passed on the stack. On function entry, the first stack parameter is at the address (SP+8).

Structures are passed via the stack.

Examples:

```
void func1( int i,  char * p, char c );    /* r3 r4   r5 */
void func2( int i1, double d, int i2 );    /* r3 r5r6 r7 */
```

Function return values

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

Return Type	Register
Arithmetic and structures 8-bit	r3
Arithmetic and structures 16-bit	r3
Arithmetic and structures 32-bit	r3
Arithmetic and structures 64-bit	r3r4
_Complex double	r3r4r5r6

Objects larger than 64 bits are returned via the stack. For structures larger than 64 bits, the first parameter register contains a pointer to the return buffer.

1.8.2. Register Usage

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

Register	Usage
r0	Register which may be modified during function linkage
r1	Stack frame pointer, always valid
r2	Small data area register (sdata2)
r3-r6	Registers used for parameter passing and return values
r7-r10	Registers used for parameter passing
r11-r12	Registers that may be modified during function linkage
r13	Small data area pointer register
r14-r31	Registers used for local variables
cr	Condition register

Register	Usage
cr0-cr7	Condition register fields, each 4 bits wide
msr	Machine status register
acc	SPE accumulator
xer	Integer exception register
lr	Link register
ctr	Count register
spefscr	Signal processing and embedded floating-point status and control register

1.8.3. Inlining Functions: inline

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

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

```
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 1.8.7, *Intrinsic Functions*.

1.8.4. VLE Instruction Support: __vle, __novle

By default VLE instructions are supported if they are available in your selected processor core. With the C compiler option **--no-vle** you can turn VLE support off. With the function qualifiers `__vle` and `__novle` you can overrule this behavior for a specific function.

Example:

```
__novle void no_vlefunc(void)
{
 // VLE instructions are not used
}

__vle void vle_func(void)
{
 // VLE instructions can be used
}
```

1.8.5. Floating-Point Unit Support: __fpu, __nofpu

By default hardware floating-point instructions are supported if a floating-point unit (FPU) is available on your selected processor core. With the C compiler option **--no-fpu** you can turn FPU support off. With the function qualifiers `__fpu` and `__nofpu` you can overrule this behavior for a specific function.

Example:

```
__nofpu void no_fpufunc(void)
{
 // hardware floating-point instructions are not used
}

__fpu void fpu_func(void)
{
 // hardware floating-point instructions can be used
}
```

1.8.6. Interrupt Functions

By default the C compiler generates normal (non-interrupt) functions. To create an interrupt function the C compiler supports a number of function attributes. They are described in the following sections.

For more information on interrupts see chapter *Interrupts and Exceptions* in the *e200zx Power Architecture Core Reference Manual*.

Defining an interrupt function

You can use the attribute `interrupt([`*vector*`] )` to declare a function as an interrupt function. You can use the `__attribute__(())` keyword for this, or you can use `__interrupt([`*vector*`] )`. With the optional *vector* argument you can bind the interrupt function to a specific vector. Without the *vector* the function is not bound to an interrupt vector. You can assign an unbound function to an interrupt vector in the linker LSL file. The linker generates sections with the vectors of the specified interrupt numbers.

For example:

```
void isr( void ) __attribute__((interrupt()))
{
    return;
}
```

Or you can use:

```
void isr( void ) __interrupt__()
{
    return;
}
```

This example creates an interrupt function, but the C compiler does not generate code to initialize the appropriate IVOR*n* SPR. This has to be done by application code.

When you use the *vector* argument, the C compiler generates code to initialize the appropriate IVOR*n* SPR. For example:

```
void isr( void ) __attribute__((interrupt(8)))

{
    return;
}
```

results into:

```
     .section  .text_vle.isr.interrupt.isr   ; interrupt function
     .global isr
     .align  16
isr: .type   func
     b       .L2
.L2:
     rfi
     .size   isr,$-isr
     .endsec

     .section  .text_vle.ivor_init, protect  ; initialization
     .align  2
     lis     r3,@ha(isr)
     la      r3,@lo(isr)(r3)
     rlwinm  r3,r3,0,16,27
     mtivor8 r3
     .endsec
```

The C compiler generates code to save and restore the used resources in the interrupt function. An interrupt function has an alignment of 16 bytes and is allocated in a code section with the following standard section name:

- `.text_vle.isr` for VLE encoded interrupt functions.
- `.text.isr` for non-VLE encoded interrupt functions.

As is the case with code sections for normal functions, the standard section name is extended by default with the module name and symbol name. You can overrule the default extension with C compiler option **--rename-sections** and/or `#pragma section`. Like for normal functions, the section type must be set to `text_vle` or `text`. See Section 1.9, *Section Naming* for details.

An interrupt function cannot return a value and cannot have parameters. It is also not possible to call an interrupt function directly from an application.

Interrupt classes

The C compiler supports the following interrupt classes:

Class	Description	Attribute	Return instruction	Save-restore registers	Mask bit
Non-critical	First-level interrupts that let the processor change program flow to handle conditions generated by external signals, errors, or unusual conditions arising from program execution or from programmable timer-related events.	base	rfi	srr0 and srr1	MSR[EE]
Critical	Critical input, watchdog timer, and debug interrupts. These interrupts can be taken during a non-critical interrupt or during regular program flow.	critical	rfci	csrr0 and csrr1	MSR[CE]
Debug	Provides a separate set of resources for the debug interrupt. Check your processor information to see whether the debug interrupt is implemented.	debug	rfdi	dsrr0 and dsrr1	MSR[DE]
Machine check	Provides a separate set of resources for the machine check interrupt. Check your processor information to see whether the machine check interrupt is implemented.	machine	rfmci	mcsrr0 and mcsrr1	MSR[ME]
Reset	An interrupt function of this class does not save and restore any resources, it also does not use a special return instruction.	reset	--	--	--

You can attach a class attribute to a function with the `__attribute__(())` keyword. For example:

```
void critical_input( void ) __attribute__((interrupt( 0 ),critical))
{
    return;
}
```

This creates a critical interrupt handler and binds the handler to vector 0:

```
   .section  .text_vle.isr.man.critical_input  ;; allocate in .text_vle.isr
   .global critical_input
   .align  16                   ;; align to 16 bytes
critical_input:  .type  func
   rfci                         ;; use appropriate return instruction
   .size   critical_input,$-critical_input
   .endsec

   .section  .text_vle.ivor_init, protect  ;; initialize appropriate IVORn SPR
   .align  2
   lis     r3,@ha(critical_input)
   la      r3,@lo(critical_input)(r3)
   rlwinm  r3,r3,0,16,27
   mtivor0 r3
   .endsec
```

When you specify an interrupt class without the `interrupt()` attribute the C compiler assumes an interrupt function of the specified class that is not bound to a vector. When you use the `interrupt()` attribute without specifying an interrupt class, a non-critical (base) class interrupt is assumed. Thus:

Declaration	Equivalent
`void isr ( void )` `__attribute__((machine));`	`void isr ( void )` `__attribute__((machine,interrupt()));`
`void isr ( void )` `__attribute__((interrupt(`[*vector*]`)));`	`void isr ( void )` `__attribute__((interrupt(`[*vector*]`),base));`

Nested interrupts

It is possible to re-enable interrupts of the same class from an interrupt handler, so that the interrupt handler itself can be interrupted again. This is called a nested interrupt function. Before the interrupts are re-enabled the save-restore registers must be preserved. For this purpose you can use the `nested` attribute. When you specify this attribute for an interrupt function, the C compiler saves and restores the appropriate save-restore registers (as listed in the table above) in the interrupt frame. The C compiler does not generate code to re-enable the interrupts. This must be done by the application itself. An example of a nested interrupt function is:

```
void system_call( void ) __attribute__((interrupt( 8 ),base,nested))
{
    return;
}
```

The generated code for this function is:

```
   .section  .text_vle.isr.man.system_call
   .global system_call
   .align  16
system_call:  .type func
   stwu    r1,-24(r1)   ;; allocate stack frame
   stw     r0,20(r1)    ;; save scratch register
   mfsrr1  r0
   stw     r0,16(r1)    ;; save srr1 register
   mfsrr0  r0
   stw     r0,12(r1)    ;; save srr0 register

;; Application can re-enable interrupt here

   lwz     r0,16(r1)
   mtsrr1  r0           ;; restore srr1 register
   lwz     r0,12(r1)
   mtsrr0  r0           ;; restore srr0 register
   lwz     r0,20(r1)    ;; restore scratch register
   addi    r1,r1,24     ;; release stack frame
   rfi
   .size   system_call,$-system_call
   .endsec
   .section .text_vle.ivor_init, protect ;; initialize appropriate IVORn SPR
   .align  2
   lis     r3,@ha(system_call)
   la      r3,@lo(system_call)(r3)
   rlwinm  r3,r3,0,16,27
   mtivor8 r3
   .endsec
```

You cannot use the `nested` attribute for non-interrupt functions.

Enable interrupts

The `enable` attribute is similar to the `nested` attribute. The only difference is that the C compiler also generates code to re-enable the interrupts of the same class. It does so right after the save-restore registers have been preserved. Right before the save-restore registers are restored, the interrupts is disabled again. The affected mask-bit in the Machine State Register (MSR) is listed in the table on interrupt classes above. Example:

```
void system_call( void ) __attribute__((interrupt( 8 ),enable))
{
    return;
}
```

The generated code for this function is:

```
    .section  .text_vle.isr.man.system_call
    .global system_call
    .align  16
system_call:  .type func
    stwu    r1,-24(r1)
    stw     r0,20(r1)
    mfsrr1  r0
    stw     r0,16(r1)    ;; preserve save-restore registers
    mfsrr0  r0
    stw     r0,12(r1)
    wrteei  1            ;; enable interrupts

;;  Application code

    wrteei  0            ;; disable interrupts
    lwz     r0,16(r1)
    mtsrr1  r0           ;; restore save-restore registers
    lwz     r0,12(r1)
    mtsrr0  r0
    lwz     r0,20(r1)
    addi    r1,r1,24
    rfi
    .size   system_call,$-system_call
    .endsec
```

You cannot use the `enable` attribute for non-interrupt functions.

Note that to avoid redundant interrupts, for some interrupts the software must take actions to clear the exception status before re-enabling interrupts. Refer to your processor manual for more information on this subject. When the exception status must be cleared first, it is not possible to use the `enable` attribute. In this case, use the `nested` attribute and re-enable interrupts in your application code once the exception status has been cleared. There are intrinsic functions available to enable/disable interrupts.

Pointers to an interrupt function

You must use the type qualifier `__isr` to create a pointer to an interrupt function. This prevents that an interrupt function is accidentally called directly from an application through a function pointer. Examples:

```
extern void isr( void ) __attribute__((debug));
extern void f( void );

void       (*fp_a)( void );
void       (*fp_b)( void );
void       (*fp_c)( void );
void __isr (*fp_isr_a)( void );
void __isr (*fp_isr_b)( void );

int main( void )
{
    fp_a = f;          /* ok, non-interrupt function, non-interrupt pointer */
    fp_isr_a = isr;    /* ok, interrupt function, interrupt pointer */
    fp_b = isr;        /* warning, interrupt function, non-interrupt pointer */
    fp_isr_b = f;      /* ok, non-interrupt function, interrupt pointer */
    fp_c = fp_isr_a;   /* warning, interrupt pointer assigned to non-interrupt
                          pointer */
    isr();             /* error, cannot call an interrupt function */
    (*fp_isr_a)();     /* error, cannot call an interrupt function */
    return 0;
}
```

Locating interrupt functions

Because the vector address of an interrupt function is built from the IVPR and IVOR*n* SPRs:

```
address = IVPR[32-47] || IVORn[48-59] || 0b0000
```

all interrupt functions use the same 16-bit IVPR prefix. Therefore, all interrupt functions must be located within the same 64 KB segment. The default linker script files contain rules to accomplish this. Because the lowest 4 bits of the address are always set to zero, the compiler aligns interrupt functions at a 16-byte boundary.

Automatic creation of the interrupt vector initialization code

When an interrupt functions is bound to a specific vector with the `interrupt()` attribute, the compiler generates a code snippet to initialize the corresponding IVOR*n* SPR. The initialization snippet(s) are placed in a section with the name: `.text_vle.invor_init`. For example:

```
void isr( void ) __attribute__((interrupt(8)))

{
    return;
}
```

results into:

```
     .section  .text_vle.ivor_init, protect  ; initialization
     .align  2
     lis     r3,@ha(isr)
     la      r3,@lo(isr)(r3)
     rlwinm  r3,r3,0,16,27
     mtivor8 r3
     .endsec
```

The default linker script files contain rules to combine the initialization snippets together with function prologue and epilogue code from the run-time library to form a function. The created function also initializes the interrupt vector prefix (IVPR) register. The name of the created function is: `__ivor_init` and it is called from the startup function: _START. The following code is an example of the generated initialization function. The example is taken from an absolute ELF file:

```
                               .section .text_vle.ivor_prologue
00000074 7c7f0ba6 __ivor_init  mtspr     0x3f,r3    ;; initialize IVPR
                               .endsec


                               .section .text_vle.ivor_init
00000078 7060e000              e_lis     r3,0
0000007c 1c630040              e_add16i  r3,r3,0x40
00000080 74630437              e_rlwinm  r3,r3,0,0x10,0x1b
00000084 7c7863a6              mtspr     0x198,r3   ;; initialize IVOR8
                               .endsec


                               .section .text_vle.ivor_init
00000088 7060e000              e_lis     r3,0
0000008c 1c630060              e_add16i  r3,r3,0x60
00000090 74630437              e_rlwinm  r3,r3,0,0x10,0x1b
00000094 7c7063a6              mtspr     0x190,r3   ;; initialize IVOR0
                               .endsec


                               .section .text_vle.ivor_epilogue
00000098 0004                  se_blr               ;; return
                               .endsec
```

It is not possible to change the section name of the initialization snippets. By default the section type for the initialization snippets is `.text_vle`. This changes into `.text` with the C compiler option **--no-vle**. Because the initialization snippets are combined into a function this option must be set to the same value for all modules in the application.

Customizing the interrupt vector initialization code

When you need to customize the interrupt vector initialization code, it is possible to create your own function with the following name and prototype:

```
void __ivor_init( unsigned long segment_address );
```

When this function is linked with your application, it will overrule the automatically created function. The startup code passes the segment address of the interrupt functions to the `__ivor_init()` function. De following example demonstrates how to create an unbound interrupt function, and use customized code to bind the function to IVOR0:

```
void isr( void ) __attribute__((interrupt()))
{
    return;
}

void __ivor_init( unsigned long segement_address )
{
    __asm( "rlwinm  r3,%0,0,16,27\n"
         "\tmtivor0 r3\n" : : "r"(isr) : "r3" );
    return;
}

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

Note that interrupt functions may be removed from your application by the linker when they are not referenced. They will reappear once the vector initialization code references the interrupt function. Alternatively, it is possible to use the `protect` attribute:

```
void isr( void ) __attribute__((interrupt(),protect));
```

1.8.7. Intrinsic Functions

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

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

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

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

The TASKING C compiler for Power Architecture recognizes the following intrinsic functions:

Intrinsics used by the compiler

__va_start

```
struct __va_list * volatile __va_start( void );
```

Used by the `va_start()` macro to initialize an object of type: `va_list`.

__va_type

```
__va_arg_type volatile __va_type( int typeid );
```

Used by `va_arg()` macro to determine the argument type parameter when calling the `__va_arg()` function.

__alloc

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

Used by the compiler to allocate space on the stack for a VLA (Variable length array). Returns a pointer to space in external memory of size bytes length. NULL if there is not enough space left.

__free

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

Used by the compiler to free the allocated space pointed to by `ptr` for a VLA (Variable length array). `ptr` must point to memory earlier allocated by a call to `__alloc()`.

User intrinsics

__get_return_address

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

Returns the return address of a function.

__get_cr

```
unsigned int * volatile __get_cr( void );
```

Return a pointer to the stack area where the condition register is saved. Can only be used together with the `push_cr` symbol attribute.

__get_gpr32

```
unsigned int * volatile __get_gpr32( void );
```

Return a pointer to the stack area where the 32-bit registers are saved. Register r0 is at offset 0. Can only be used together with the `push_gpr32` symbol attribute.

__get_srr

```
unsigned int * volatile __get_srr( void );
```

Return a pointer to the stack area where the save-restore registers are saved. Can only be used together with the `push_srr` or `nested` symbol attribute.

__nop

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

Generate a NOP instruction.

__sc

```
void volatile __sc( void );
```

Generate a system call (SC) instruction.

__illegal

```
void volatile __illegal( void );
```

Generate an illegal instruction.

__mfmsr

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

Return the contents of the machine state register.

__mtmsr

```
void volatile __mtmsr( unsigned int value );
```

Move `value` to the machine state register.

__wrtee

```
void volatile __wrtee( unsigned int value );
```

Move `value` to MSR[EE] using the `wrtee` instruction.

__wrteei

```
void volatile __wrteei( _Bool state );
```

Move `state` to MSR[EE] using the `wrteei` instruction. The argument must be a constant value.

__tlbre

```
void volatile __tlbre( void );
```

Generate a `tlbre` instruction.

__tlbwe

```
void volatile __tlbwe( void );
```

Generate a `tlbwe` instruction.

__tlbsx

```
void volatile __tlbsx( void * ea );
```

Generate a `tlbsx` instruction.

__tlbivax

```
void volatile __tlbivax( void * ea );
```

Generate a `tlbivax` instruction.

__tlbsync

```
void volatile __tlbsync( void );
```

Generate a `tlbsync` instruction.

__msync

```
void volatile __msync( void );
```

Generate a `msync` instruction.

__isync

```
void volatile __isync( void );
```

Generate a `isync` instruction.

__mtmas

```
void volatile __mtmas[0..4|6]( unsigned int value );
```

Move value to one of the MMU assist registers: MAS[0..4|6].

__mfmas

```
unsigned int volatile __mfmas[0..4|6]( void );
```

Read value from one of the MMU assist registers: MAS[0..4|6].

__fabsf

```
float __fabsf( float value );
```

Return the absolute value of a single precision floating-point number.

__fabs

```
double __fabs( double value );
```

Return the absolute value of a double precision floating-point number.

1.9. Section Naming

By default the compiler generates section names that start with a dot ('.') and the section type, extended with the module name and the name of the symbol that is allocated in the section. Each component is separated by a dot ('.'):

```
  .type.module-name.symbol-name
```

The section types are: `rodata`, `data`, `bss`, `sdata`, `sbss`, `sdata0`, `sbss0`, `sdata2`, `sbss2`, `text`, `text_vle`.

Section names are case sensitive. By default, the sections are not concatenated by the linker. This means that multiple sections with the same name may exist. At link time sections with different attributes can be selected on their attributes. The linker may remove unreferenced sections from the application.

You can change the default section name extension (using the module name and symbol name) with a pragma or with a command line option.

```
--rename-sections[=[type[.attribute]=][format_string]],...
```

```
#pragma section [[type[.attribute]=|whitespace][format_string]],...
```

Note that the pragma has a slightly different syntax because the equal sign ('=') may be replaced by whitespace.

With the section *type* and *attribute* you select which sections will be renamed. When the type and attributes of a section match, the section name will be set to a dot ('.') and the section type, extended with the specified *format string*. The compiler will add a dot ('.') between the section type and the format string automatically.

The following attributes are allowed: `init`, `noclear`, `romdata`.

When you specify an optional attribute, only sections that have the attribute will be renamed. When you do not specify an attribute, only sections that do not have any of the listed attributes will be renamed. Note that the listed attributes are mutually exclusive; if a section uses one of the attributes, the other attributes will not be used.

When the type and attribute are omitted, or type "**all**" is used, all sections will be renamed.

The format string can contain characters and may contain the following format specifiers:

`{attrib}`	section attributes, separated by dots
`{module}`	module name
`{name}`	object name, name of variable or function

Within format specifier expansions, dots ('.') will be replaced by dollars ('$').

When the format string is omitted, a section will have a name that consist of a dot ('.') and the section type only. Note that `#pragma section ,,data=special` will give all sections a name consisting of just the section type, except for data sections, which will be named "`.data.special`". The reason for this is that the double comma is interpreted as: **--rename-sections** or `#pragma section`, without arguments.

Some examples (file `test.c`):

```
#pragma section bss={module}
__data int x;
/* Section name: .bss.test */

#pragma section sdata=myname.{name}
__sdata int status=1;
/* Section name: .sdata.myname.status */
```

With `#pragma endsection` the naming convention of the previous level is restored, while with `#pragma section default` the command line state of the **--rename-sections** option is restored, or when the **--rename-sections** option was not used, the default section naming convention is restored. Nesting of pragma section/endsection pairs will save the status of the previous level.

Examples (file `example.c`)

```
char a;        // allocated in '.sbss.example.a'
#pragma section sbss=Mysect1
char b;        // allocated in '.sbss.Mysect1'
#pragma section sbss=Mysect2
char c;        // allocated in '.sbss.Mysect2'
#pragma endsection
char d;        // allocated in '.sbss.Mysect1'
#pragma endsection
char e;        // allocated in '.sbss.example.e'
```

Chapter 2. Assembly Language

This chapter describes the most important aspects of the TASKING assembly language for Power Architecture. For a complete overview of the architecture you are using, refer to the target's *Architecture Core Reference Manual*. For a description of the assembly instruction set, refer to the *Programmer's Reference Manual for Freescale Embedded Processors*.

2.1. 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
LAB1      ; This label has to start at the beginning
          ; of a line
1: b 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 2.3, *Operands of an Assembly Instruction*. The instructions are described in the *Programmer's Reference Manual for Freescale Embedded Processors*. The instruction can also be a so-called 'generic instruction'. Generic instructions are pseudo instructions (no instructions from the instruction set). Depending on the situation in which a generic instruction is used, the assembler replaces the generic instruction with appropriate real assembly instruction(s). For a complete list, see Section 2.11, *Generic Instructions*.
directive	With directives you can control the assembler from within the assembly source. Except for preprocessing directives, these must not start in the first column. Directives are described in Section 2.9, *Assembler Directives*.
macro_call	A call to a previously defined macro. It must not start in the first column. See Section 2.10, *Macro Operations*.
comment	Comment, preceded by a ; (semicolon).

You can use empty lines or lines with only comments.

2.2. 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 2.6.3, *Expression Operators*. Other special assembler characters are:

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

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

2.3. 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 2.4, *Symbol Names*. Symbols can also occur in expressions.
register	Any valid register as listed in Section 2.5, *Registers*.
expression	Any valid expression as described in Section 2.6, *Assembly Expressions*.
address	A combination of *expression*, *register* and *symbol*.

Addressing modes

The Power Architecture assembly language has several addressing modes. These are described in detail in the *Programmer's Reference Manual for Freescale Embedded Processors*.

2.4. 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 2.4.1, *Predefined Preprocessor Symbols*.

Labels

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

Reserved symbols

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

Examples

Valid symbol names:

```
loop_1
ENTRY
a_B_c
_aBC
```

Invalid symbol names:

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

2.4.1. 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
__ASPPC__	Identifies the assembler. You can use this symbol to flag parts of the source which must be recognized by the **asppc** assembler only. It expands to 1.
__BUILD__	Identifies the build number of the assembler, composed of decimal digits for the build number, three digits for the major branch number and three digits for the minor branch number. For example, if you use build 1.22.1 of the assembler, __BUILD__ expands to 1022001. If there is no branch number, the branch digits expand to zero. For example, build 127 results in 127000000.
__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('__ASPPC__')
  ; this part is only for the asppc assembler
...
.endif
```

2.5. Registers

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

```
  r0   .. r31    (32 general purpose registers)
  fr0  .. fr31   (32 floating-point registers)
```

```
fsl0 .. fsl31  (32 FSL copprocessor registers)
cr0  .. cr7    (8 control registers)
```

2.5.1. Special Purpose Registers (SPRs)

The e200*zx* Power Architecture™ Core Reference Manuals contain a list of all special purpose registers. Some SPRs behave differently on different cores. The deviations are listed below:

- SPR PIR with value 286 is writable for the cores e200z4 and e200z7. For the other cores, this SPR is read only.
- SPRs TBU and TBL have multiple values. However values 268 and 269 respectively must be used for reading the SPR and values 284 and 285 respectively must be used for writing the SPR.
- SPRs ALTCTXCR and IAC8 share the same value: 568. Fortunately, the ALTCTXCR is only used for core e200z6, and IAC8 is used for cores e200z4 and e200z7. So this will not lead to a conflict.
- SPRs SPRG[4567] have multiple values: 276/260, 277/261, 277/262 and 279/263. The low numbers are intended to be used in user mode, and these SPRs are read only. The high numbers are intended to be used in supervisor mode and are writable.

The last item gives some problems. To choose the right value, the assembler should be aware of the current mode (user or supervisor). However it is not. The value of the SPRs need to be known. But not necessarily the mnemonics. Therefore, the TASKING assembler supports new mnemonics for SPRG[4567] in supervisor mode: S_SPRG[4567]. This is how it is implemented in the assembler:

- SPRG[4567] get the values 260, 261, 262 and 263.
- S_SPRG[4567] get the values 276, 277, 277 and 279.

This also works with the mt*spr* and mf*spr* instruction mnemonics.

2.6. 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.

2.6.1. Numeric Constants

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

Base	Description	Example

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

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

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

2.6.2. 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'              ; (0x41424344)
'''79'              ; to enclose a quote double it
"A\"BC"             ; or to enclose a quote escape it
'AB'+1              ; (0x4143) string used in expression
''                  ; null string
.DW 'abcdef'        ; (0x61626364) 'ef' are ignored
                    ; warning: string value truncated
'abc'++'de'         ; you can concatenate
                    ; two strings with the '++' operator.
                    ; This results in 'abcde'
['TASKING',0,4]     ; results in the substring 'TASK'
```

2.6.3. 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.

Type	Operator	Name	Description
	%	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. 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. Use tests for equality involving floating-point values with caution, since rounding errors could cause unexpected results.
	<=	less than or equal	
	>	greater than	
	>=	greater than or equal	
	==	equal	
	!=	not equal	
Bitwise	&	AND	Integer only. Yields the bitwise AND function of its operand.
	|	OR	Integer only. Yields the bitwise OR function of its operand.
	^	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.

2.7. 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 5.7.7, *The Section Layout Definition: Locating Sections*.

Section definition

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

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

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

Examples

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

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

2.8. 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
`@BIGENDIAN()`	Test if assembler generates code for big-endian mode
`@CNT()`	Return number of macro arguments
`@CPU('`*architecture*`')`	Test if current CPU matches *architecture*
`@DEFINED('`*symbol*`' |` *symbol*`)`	Test whether *symbol* exists
`@HA(`*expr*`)`	Returns upper 16 bits of expression value, adjusted for signed addition
`@HI(`*expr*`)`	Returns upper 16 bits of expression value
`@LO(`*expr*`)`	Returns lower 16 bits of expression value
`@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
`@RELSDA(`*symbol*`)`	Offset of *symbol* in small data area (SDA)
`@RELSDAHA(`*symbol*`)`	Upper 16 bits of offset of *symbol* in small data area (SDA), adjusted for signed addition
`@RELSDAHI(`*symbol*`)`	Upper 16 bits of offset of *symbol* in small data area (SDA)
`@RELSDALO(`*symbol*`)`	Lower 16 bits of offset of *symbol* in small data area (SDA)
`@SDA21(`*symbol*`)`	Offset of *symbol* in small data area (SDA) and fill in base address register
`@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*

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?
```

@BIGENDIAN()

Returns 1 if the assembler generates code for big-endian mode (this is the default), returns 0 if the assembler generates code for little-endian mode.

@CNT()

Returns the number of macro arguments of the current macro expansion as an integer. If you use this function when macro expansion is not active, the assembler issues a warning.

Example:

```
ARGCOUNT .SET @CNT() ; reserve argument count
```

@CPU('*architecture*')

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

Example:

```
.IF @CPU('e200z0')    ; true if you specified option --core=e200z0
 ... ; code for the e200z0
.ELIF @CPU('e200z6')  ; true if you specified option --core=e200z6
 ... ; code for the e200z6
.ELSE
 ... ; code for other architectures
.ENDIF
```

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

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

Example:

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

@HA(*expression*)

Returns the upper 16 bits of a value, adjusted for a signed addition. `@HA(expression)` is equivalent to `(((expression+0x800)>>16) & 0xffff)`. *expression* can be any relocatable or absolute expression.

Example:

```
;; initialize the stack pointer
lis r3,@ha(_lc_bs)
la  r4,@lo(_lc_bs)(r3)
```

@HI(*expression*)

Returns the upper 16 bits of a value. `@HI(expression)` is equivalent to `((expression>>16) & 0xffff)`. *expression* can be any relocatable or absolute expression.

@LO(*expression*)

Returns the lower 16 bits of a value. `@LO(expression)` is equivalent to `(expression & 0xffff)`. *expression* can be any relocatable or absolute expression.

@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).

@RELSDA(*symbol*)

Returns the offset of *symbol* in the small data area (SDA) in which *symbol* will be located.

@RELSDAHA(*symbol*)

Returns the upper 16 bits, adjusted for a signed addition, of the offset of *symbol* in the small data area (SDA) in which *symbol* will be located.

@RELSDAHI(*symbol*)

Returns the upper 16 bits of the offset of *symbol* in the small data area (SDA) in which *symbol* will be located.

@RELSDALO(*symbol*)

Returns the lower 16 bits of the offset of *symbol* in the small data area (SDA) in which *symbol* will be located.

@SDA21(*symbol*)

Same as `@RELSDA()`, but also fills in the base address register corresponding to the SDA.

@STRCAT(*string1*,*string2*)

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

Example:

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

@STRCMP(*string1*,*string2*)

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

<0 if *string1* < *string2*

0 if *string1* == *string2*

>0 if *string1* > *string2*

Example:

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

@STRLEN(*string*)

Returns the length of *string* as an integer.

Example:

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

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

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

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

Example:

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

2.9. Assembler Directives

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

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

  The following directives fall under this group:

  - Assembly control directives
  - Symbol definition and section directives
  - Data definition / Storage allocation directives
  - High Level Language (HLL) directives

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

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

Each assembler directive has its own syntax. Some assembler directives can be preceded with a label. If you do not precede an assembler directive with a label, you must use white space instead (spaces or tabs). You can use assembler directives in the assembly code as pseudo instructions. The assembler recognizes both upper and lower case for directives.

2.9.1. Overview of Assembler Directives

The following tables provide an overview of all assembler directives. For a detailed description of these directives, refer to Section 2.9.2, *Detailed Description of Assembler Directives*.

Overview of assembly control directives

Directive	Description
`.END`	Indicates the end of an assembly module
`.INCLUDE`	Include file
`.MESSAGE`	Programmer generated message

Overview of symbol definition and section directives

Directive	Description
`.EQU`	Set permanent value to a symbol
`.EXTERN`	Import global section symbol
`.GLOBAL`	Declare global section symbol
`.RESUME`	Resume a previously defined section
`.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
`.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)
`.DS`, `.DSB`, `.DSH`, `.DSW`, `.DSD`	Define storage

Overview of macro preprocessor directives

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

Overview of listing control directives

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

Overview of HLL directives

Directive	Description
`.CALLS`	Pass call tree information and/or stack usage information
`.MISRAC`	Pass MISRA-C information

2.9.2. Detailed Description of Assembler Directives

.ALIGN

Syntax

```
.ALIGN expression
```

Description

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

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

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

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

A label is not allowed before this directive.

Example

```
.SECTION  .text
.ALIGN 2     ; the assembler aligns
instruction  ; this instruction at 2 MAUs and
             ; fills the 'gap' with NOP instructions.
.ENDSEC

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

.BREAK

Syntax

```
.BREAK
```

Description

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

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

The assembler does not allow a label with this directive.

Example

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

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

Syntax

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

Description

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

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

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

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

You cannot initialize of a block of memory in sections with prefix `.sbss` or `.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. A function can use multiple stacks.

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

This directive is generated by the C compiler. 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.

.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 0x414200000044 , the 'C' is truncated
WTBL: .DW 'ABC'         ; results in 0x00414243
```

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

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

The assembler does not allow a label with this directive.

Example

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

.EQU

Syntax

`symbol` **`.EQU`** *`expression`*

Description

With the `.EQU` directive you assign the value of *expression* to *symbol* permanently. The expression can be relative or absolute. Once defined, you cannot redefine the symbol. With the `.GLOBAL` directive you can declare the symbol global.

Example

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

```
MYSYMBOL .EQU  0x4000
```

You cannot redefine the symbol `MYSYMBOL` after this.

Related Information

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

.EXTERN

Syntax

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

Description

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

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

A label is not allowed with this directive.

Example

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

Related Information

`.GLOBAL` (Declare global section symbol)

.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)

.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
```

```
  ... ; 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
```

Related Information

Assembler option **--define** (Define preprocessor macro)

.INCLUDE

Syntax

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

Description

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

The string specifies the filename of the file to be included. The filename must be compatible with the operating system (forward/backward slashes) and can contain a directory specification. If you omit a filename extension, the assembler assumes the extension `.asm`.

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

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

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

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

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

4. The default `include` directory in the installation directory.

The assembler does not allow a label with this directive.

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

Example

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

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

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

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

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

Related Information

Assembler option **--include-directory** (Add directory to include file search path)

.LIST, .NOLIST

Syntax

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

Description

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

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

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

Example

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

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

The assembler generates a list file with the following lines:

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

Related Information

Assembler option **--list-file** (Generate list file)

.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:

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

The macro call:

```
    .section .data
macro_a 2,3
    .endsec
```

The macro expands as follows:

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

Related Information

Section 2.10, *Macro Operations*

`.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
```

.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 3.8.2, *C Code Checking: MISRA-C*

C compiler option **--misrac**

.PAGE

Syntax

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

Default

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

Description

If you generate a list file with the assembler option **--list-file**, you can use the directive `.PAGE` 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.
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).
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*.

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

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

Example

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

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

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

Related Information

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

Assembler option **--list-file**

.REPEAT, .ENDREP

Syntax

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

Description

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

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

Example

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

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

Related Information

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

.RESUME

Syntax

```
.RESUME name[,attribute]...
```

Description

With the `.SECTION` you always start a new section. With the `.RESUME` directive you can reactivate a previously defined section. See the `.SECTION` directive for a list of available section attributes. If you omit the attribute, the previously defined section with the same name is reactivated (ignoring the attribute(s)). If you specify an attribute you reactivate the section with that same attribute.

Example

```
.SECTION .text              ; First .text section
 ...
.SECTION .data              ; First .data section
 ...
.SECTION .text              ; Second .text section
 ...
.SECTION .data, at(0x0)     ; Second .data section
 ...
.RESUME .text               ; Resume in the second .text section
 ...
.RESUME .data               ; Resume in the first .data section
 ...
.RESUME .data, at(0x0)      ; Resume in the second .data section
```

Related Information

`.SECTION` (Start a new section)

.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.

To resume a previously defined section, use the `.RESUME` directive.

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

You can use the following predefined section names:

Section name	Description	Section type
.text_vle	Code sections with support for VLE instructions	code_vle
.text	Code sections (VLE instructions not allowed)	code
.init	Code section with initialization code	code
.data	Initialized data	data
.sdata	Initialized data in read-write small data area (R13 based)	data
.sdata2	Initialized data in read-only small data area (R2 based)	data
.bss	Uninitialized data (cleared)	data
.sbss	Uninitialized data in read-write small data area (cleared)	data
.sbss2	Uninitialized data in read-only small data area (cleared)	data
.rodata	ROM data (constants)	data
.debug	Debug sections	debug

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

Attribute	Description	Allowed on type
ALIGN(*value*)	Align the section to the value specified. *value* must be a power of two. The default alignment of CODE_VLE sections is two bytes. The default alignment of CODE sections is four bytes.	CODE, CODE_VLE, DATA
AT(*address*)	Locate the section at the given *address*.	CODE, CODE_VLE, DATA
CLEAR	Sections are zeroed at startup.	DATA

Attribute	**Description**	**Allowed on type**
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, CODE_VLE, DATA, DEBUG
INIT	Defines that the section contains initialization data, which is copied from ROM to RAM at program startup.	CODE, CODE_VLE, 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.	DATA
NOCLEAR	Sections are not zeroed at startup. This is a default attribute for data sections. This attribute is only useful with BSS sections, which are cleared at startup by default.	DATA
NOINIT	Defines that the section contains no initialization data.	CODE, CODE_VLE, DATA
PROTECT	Tells the linker to exclude a section from unreferenced section removal and duplicate section removal.	CODE, CODE_VLE, DATA
ROMDATA	Section contains data to be placed in ROM. This ROM area is not executable.	DATA

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

Example

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

.SECTION .sdata2, romdata   ; Declare an .sdata2 section in ROM
   ;;
.ENDSEC

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

Related Information

`.RESUME` (Resume a previously defined section)

.SET

Syntax

```
symbol  .SET  expression

        .SET  symbol  expression
```

Description

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

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

Example

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

Related Information

`.EQU` (Set permanent value to a symbol)

.SIZE

Syntax

```
.SIZE  symbol,expression
```

Description

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

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

Example

```
        .section        .text_vle
        .global main
        .align  2
; Function main
main:   .type   func
         ;
        .SIZE   main,$-main
        .endsec
```

Related Information

`.TYPE` (Set symbol type)

.SOURCE

Syntax

.SOURCE *string*

Description

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

Example

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

.TITLE

Syntax

```
.TITLE ["string"]
```

Default

```
.TITLE ""
```

Description

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

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

The `.TITLE` directive itself will not be printed in the source listing.

If the page width is too small for the title to fit in the header, it will be truncated.

Example

```
.TITLE  "This is the title"
```

Related Information

`.PAGE` (Format the assembler list file)

Assembler option **--list-file**

.TYPE

Syntax

`symbol` **`.TYPE`** *`typeid`*

Description

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

`FUNC`	The symbol is associated with a function or other executable code.
`OBJECT`	The symbol is associated with an object such as a variable, an array, or a structure.
`FILE`	The symbol name represents the filename of the compilation unit.

Labels in code sections have the default type `FUNC`. Labels in data sections have the default type `OBJECT`.

Example

```
Afunc:  .type   func
```

Related Information

`.SIZE` (Set symbol size)

.UNDEF

Syntax

```
.UNDEF symbol
```

Description

With the `.UNDEF` directive you can undefine a substitution string that was previously defined with the `.DEFINE` directive. The substitution string associated with *symbol* is released, and *symbol* will no longer represent a valid `.DEFINE` substitution or macro.

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

The assembler does not allow a label with this directive.

Example

The following example undefines the `LEN` substitution string that was previously defined with the `.DEFINE` directive:

```
.UNDEF LEN
```

Related Information

`.DEFINE` (Define a substitution string)

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

.WEAK

Syntax

```
.WEAK symbol[,symbol]...
```

Description

With the `.WEAK` directive you mark one or more symbols as 'weak'. The *symbol* can be defined in the same module with the `.GLOBAL` directive or the `.EXTERN` directive. If the symbol does not already exist, it will be created.

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

You can overrule a weak definition with a `.GLOBAL` definition in another module. The linker will not complain about the duplicate definition, and ignore the weak definition.

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

Example

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

Related Information

`.EXTERN` (Import global section symbol)

`.GLOBAL` (Declare global section symbol)

2.10. 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.

2.10.1. 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.

2.10.2. 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.

2.10.3. 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
   lis   r\reg,val
   .ENDM
```

The macro is called as follows:

```
MAC_A 3,1
```

The macro expands as follows:

```
   lis r3,1
```

The macro preprocessor substitutes the character '3' 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:

```
   lis 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 3,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
   lis   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:   lwz    r4,@relsda(^addr)(r13)
      .ENDM
```

The macro is called as follows:

```
LOCAL:
       INIT LOCAL
```

The macro expands as:

```
LOCAL__M_L000001: lwz   r4,@relsda(LOCAL)(r13)
```

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

```
LOCAL__M_L000001: lwz   r4,@relsda(LOCAL__M_L000001)(r13)
```

2.11. Generic Instructions

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

You can find a complete list of generic instructions for the Power Architecture in the embedded processor manual.

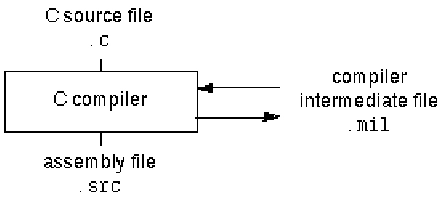
Chapter 3. Using the C Compiler

This chapter describes the compilation process and explains how to call the C compiler.

The TASKING VX-toolset for Power Architecture has a make utility to build your entire embedded project, from C source till the final ELF/DWARF object file which serves as input for a debugger.

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 6.1, *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 7.1, *C Compiler Options*. Finally, a few important basic tasks are described, such as including the C startup code and performing various optimizations.

3.1. Compilation Process

During the compilation of a C program, the C compiler runs through a number of phases that are divided into two parts: *frontend* and *backend*.

The backend part is not called for each C statement, but starts after a complete C module or set of modules has been processed by the frontend (in memory). This allows better optimization.

The C compiler requires only one pass over the input file which results in relative fast compilation.

Frontend phases

1. The preprocessor phase:

   The preprocessor includes files and substitutes macros by C source. It uses only string manipulations on the C source. The syntax for the preprocessor is independent of the C syntax but is also described in the ISO/IEC 9899:1999(E) standard.

2. The scanner phase:

   The scanner converts the preprocessor output to a stream of tokens.

3. The parser phase:

The tokens are fed to a parser for the C grammar. The parser performs a syntactic and semantic analysis of the program, and generates an intermediate representation of the program. This code is called MIL (Medium level Intermediate Language).

4. The frontend optimization phase:

   Target processor independent optimizations are performed by transforming the intermediate code.

Backend phases

1. Instruction selector phase:

   This phase reads the MIL input and translates it into Low level Intermediate Language (LIL). The LIL objects correspond to a processor instruction, with an opcode, operands and information used within the C compiler.

2. Peephole optimizer/instruction scheduler/software pipelining phase:

   This phase replaces instruction sequences by equivalent but faster and/or shorter sequences, rearranges instructions and deletes unnecessary instructions.

3. Register allocator phase:

   This phase chooses a physical register to use for each virtual register.

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.

3.2. Calling the C Compiler

Invocation syntax on the command line (Windows Command Prompt):

```
cppc [ [option]... [file]... ]...
```

The input *file* must be a C source file. You can find a detailed description of all C compiler options in Section 7.1, *C Compiler Options*.

3.3. The C Startup Code

You need the run-time startup code to build an executable application. The startup code consists of the following components:

- *Initialization code*. This code is executed when the program is initiated and before the function `main()` is called. It initializes the processor's registers and the application C variables.
- *Exit code*. This controls the close down of the application after the program's main function terminates.
- *Trap vector table*. This contains default trap vectors. See also Section 1.8.6, *Interrupt Functions*.

Hardware configuration

To configure the hardware the startup code calls a number of functions:

- `__system_init()`
- `__ivor_init()`
- `__spe_init( unsigned int ivpr )`

For all functions there are default versions present in the libraries. By adding one or more of the above functions to your application it is possible to overrule the default library functions and customize the hardware setup. Please refer to the source code of the default versions in the libraries for details about the implementation. The prototypes for these functions can be found in the `cstart.h` header file.

To add the C startup code to your project

The startup code is present in the files `lib/src/cstart.asm` and `include/cstart.h`. You can make copies of these files and add them to your project.

3.4. 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 with the 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 `CPPCINC`.
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:

```
cppc -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 `CPPCINC` 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 `CPPCINC` and then in the default `include` directory.

3.5. 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.

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 3.6, *Compiler Optimizations*.

Invocation syntax on the command line (Windows Command Prompt)

The invocation syntax on the command line is:

```
cppc -g file.c
```

3.6. Compiler Optimizations

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

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.

3.6.1. Generic Optimizations (frontend)

Common subexpression elimination (CSE) (option -Oc/-OC)

The compiler detects repeated use of the same (sub-)expression. Such a "common" expression is replaced by a variable that is initialized with the value of the expression to avoid recomputation. This method is called common subexpression elimination (CSE).

Expression simplification (option -Oe/-OE)

Multiplication by 0 or 1 and additions or subtractions of 0 are removed. Such useless expressions may be introduced by macros or by the compiler itself (for example, array subscripting).

Constant propagation (option -Op/-OP)

A variable with a known value is replaced by that value.

Automatic function inlining (option -Oi/-OI)

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

Control flow simplification (option -Of/-OF)

A number of techniques to simplify the flow of the program by removing unnecessary code and reducing the number of jumps. For example:

- *Switch optimization*: A number of optimizations of a switch statement are performed, such as removing redundant case labels or even removing an entire switch.
- *Jump chaining*: A (conditional) jump to a label which is immediately followed by an unconditional jump may be replaced by a jump to the destination label of the second jump. This optimization speeds up execution.
- *Conditional jump reversal*: A conditional jump over an unconditional jump is transformed into one conditional jump with the jump condition reversed. This reduces both the code size and the execution time.
- *Dead code elimination*: Code that is never reached, is removed. The compiler generates a warning messages because this may indicate a coding error.

Subscript strength reduction (option -Os/-OS)

An array or pointer subscripted with a loop iterator variable (or a simple linear function of the iterator variable), is replaced by the dereference of a pointer that is updated whenever the iterator is updated.

Loop transformations (option -Ol/-OL)

Transform a loop with the entry point at the bottom, to a loop with the entry point at the top. This enables constant propagation in the initial loop test and code motion of loop invariant code by the CSE optimization.

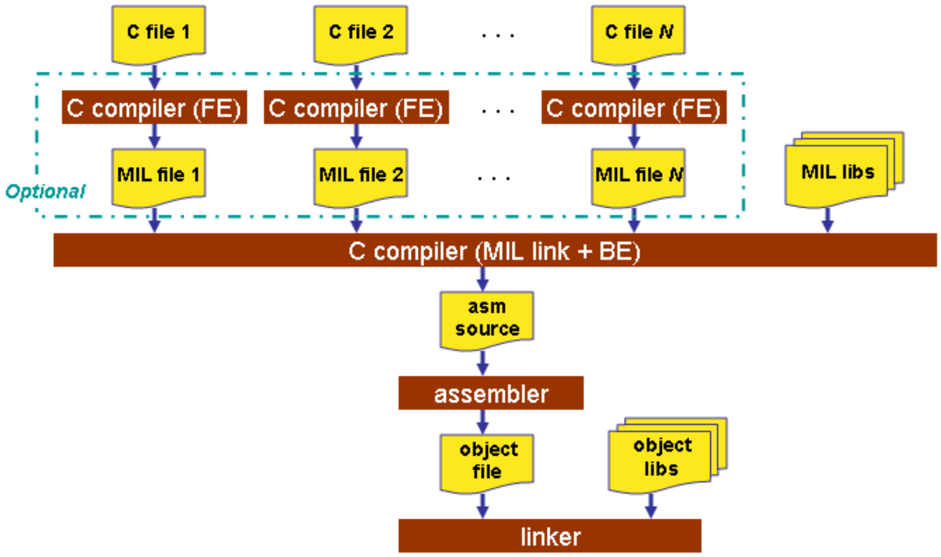
Forward store (option -Oo/-OO)

A temporary variable is used to cache multiple assignments (stores) to the same non-automatic variable.

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

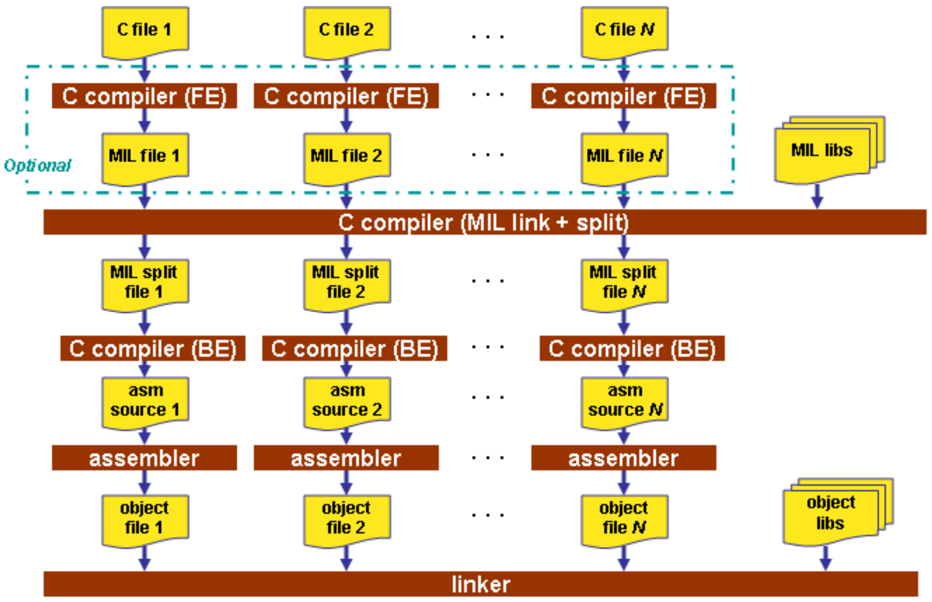
The frontend phase performs its optimizations on the MIL code. When all C modules and/or MIL modules of an application are given to the C compiler in a single invocation, the C compiler will link MIL code of the modules to a complete application automatically. Next, the frontend will run its optimizations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

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



MIL splitting (option --mil-split)

When you specify that the C compiler has to use MIL splitting, the C compiler will first link the application at MIL level as described above. However, after rerunning the optimizations the MIL code is not passed on to the backend. Instead the frontend writes a `.ms` file for each input module. 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 3.7, *Influencing the Build Time*.

3.6.2. Core Specific Optimizations (backend)

Coalescer (option -Oa/-OA)

The coalescer seeks for possibilities to reduce the number of moves (MOV instruction) by smart use of registers. This optimizes both speed and code size.

Interprocedural register optimization (option -Ob/-OB)

The generated assembly code is improved by replacing instruction sequences by equivalent but faster and/or shorter sequences, or by deleting unnecessary instructions.

Peephole optimizations (option -Oy/-OY)

The generated assembly code is improved by replacing instruction sequences by equivalent but faster and/or shorter sequences, or by deleting unnecessary instructions.

Instruction Scheduler (option -Ok/-OK)

The instruction scheduler is a backend optimization that acts upon the generated instructions. When two instructions need the same machine resource - like a bus, register or functional unit - at the same time, they suffer a *structural hazard*, which stalls the pipeline. This optimization tries to rearrange instructions to avoid structural hazards, for example by inserting another non-related instruction.

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

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

Unroll small loops (option -Ou/-OU)

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

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

Compaction is the opposite of inlining functions: chunks of code that occur more than once, are transformed into a function. This reduces code size at the cost of execution speed. The size of the chunks of code to be inlined depends on the setting of the C compiler option **--tradeoff** (**-t**). See the subsection **Code Compaction** in Section 3.6.3, *Optimize for Size or Speed*.

Generic assembly optimizations (option -Og/-OG)

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

3.6.3. Optimize for Size or Speed

You can tell the compiler to focus on execution speed or code size during optimizations. You can do this by specifying a size/speed trade-off level from 0 (speed) to 4 (size). This trade-off does not turn optimization phases on or off. Instead, its level is a weight factor that is used in the different optimization phases to influence the heuristics. The higher the level, the more the compiler focusses on code size optimization. To choose a trade-off value read the description below about which optimizations are affected and the impact of the different trade-off values.

Note that the trade-off settings are directions and there is no guarantee that these are followed. The compiler may decide to generate different code if it assessed that this would improve the result.

To specify the size/speed trade-off optimization level use 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.

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	Optimize loops for size/speed
0	no	speed
1	yes	speed
2	yes	speed
3	yes	size
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 **--tradeoff=4**) is:

```
        lwz    r0,@relsda(a)(r13)
        b      .L2                   ;; unconditional jump to loop test at bottom
.L3:
        addi   r0,r0,1
        addi   r4,r4,1
.L2:                                 ;; loop entry point
        cmp    cr0,0,r4,r3
        blt    cr0,.L3
        stw    r0,@relsda(a)(r13)
        blr
```

Coded as a top loop (compiled with **--tradeoff=0**) is:

```
        subf.  r4,r4,r3
        lwz    r0,@relsda(a)(r13)  ;; test for at least one loop iteration
        ble    cr0,.L2             ;; can be omitted when number of iterations is known
        mtctr  r4
.L3:                               ;; loop entry point
        addi   r0,r0,1
        bdnz    .L3
.L2:
        stw     r0,@relsda(a)(r13)
        blr
```

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	100	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.

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**

3.7. 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.

MIL Linking

With MIL linking 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 are:

- **MIL linking (--mil)**

  With 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.

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

  - Without this option, 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.

  - When this option is set, 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 option enabled. Although the MIL stream is split in separate modules after MIL linking, it still may happen that modifying a single C source file results in multiple MIL files to be compiled. This is a natural result of global optimizations, where the code generated for multiple modules was affected by the change.

If you are optimizing fully for speed, it is recommended to use MIL splitting instead of just MIL linking.

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.

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.

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.

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

Parallel Build

The make utility **amk**, 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.

3.8. 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.

3.8.1. 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 12, *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 12, *CERT C Secure Coding Standard*.

To apply CERT C code checking to your application

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

```
cppc --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.

3.8.2. C Code Checking: MISRA-C

The C programming language is a standard for high level language programming in embedded systems, yet it is considered somewhat unsuitable for programming safety-related applications. Through enhanced code checking and strict enforcement of best practice programming rules, TASKING MISRA-C code checking helps you to produce more robust code.

MISRA-C specifies a subset of the C programming language which is intended to be suitable for embedded automotive systems. It consists of a set of rules, defined in *MISRA-C:2004, Guidelines for the Use of the C Language in Critical Systems* (Motor Industry Research Association (MIRA), 2004).

The compiler also supports MISRA-C:1998, the first version of MISRA-C. You can select this version with the following C compiler option:

```
--misrac-version=1998
```

For a complete overview of all MISRA-C rules, see Chapter 13, *MISRA-C Rules*.

Implementation issues

The MISRA-C implementation in the compiler supports nearly all rules. Only a few rules are not supported because they address documentation, run-time behavior, or other issues that cannot be checked by static source code inspection, or because they require an application-wide overview.

During compilation of the code, violations of the enabled MISRA-C rules are indicated with error messages and the build process is halted.

MISRA-C rules are divided in required rules and advisory rules. If rules are violated, errors are generated causing the compiler to stop. With the following options warnings, instead of errors, are generated for either or both the required rules and the advisory rules:

```
--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

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

```
cppc --misrac={all | number [-number],...]
```

3.9. C Compiler Error Messages

The C compiler reports the following types of error messages.

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 with 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

On the command line you can use the C compiler option **--diag** to see an explanation of a diagnostic message:

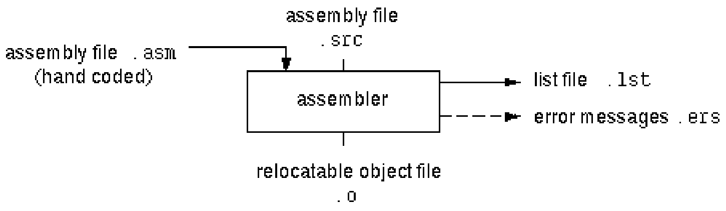
```
cppc --diag=[format:]{all | number,...]
```

Chapter 4. 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 7.2, *Assembler Options*.
- The various assembler optimizations.
- How to generate a list file.
- Types of assembler messages.

4.1. Assembly Process

The assembler generates relocatable output files with the extension `.o`. These files serve as input for the linker.

Phases of the assembly process

- Parsing of the source file: preprocessing of assembler directives and checking of the syntax of instructions
- Instruction grouping and reordering
- Optimization (instruction size and generic instructions)
- Generation of the relocatable object file and optionally a list file

The assembler integrates file inclusion and macro facilities. See Section 2.10, *Macro Operations* for more information.

4.2. Calling the Assembler

The TASKING VX-toolset for Power Architecture has a make utility to build your entire project and you can use the control program to invoke the tools in the toolset. It also possible to invoke all tools individually.

Invocation syntax on the command line (Windows Command Prompt):

```
asppc [ [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 7.2, *Assembler Options*.

4.3. 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 with the 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 `ASPPCINC`.
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:

```
asppc -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 `ASPPCINC` and then in the default `include` directory.

4.4. Assembler Optimizations

The assembler can perform various optimizations that you can enable or disable. You can enable or disable the optimizations described below. The command line option for each optimization is given in brackets.

Allow generic instructions (option -Og/-OG)

When this option is enabled, you can use generic instructions in your assembly source. The assembler tries to replace instructions by faster or smaller instructions.

By default this option is enabled. Because shorter instructions may influence the number of cycles, you may want to disable this option when you have written timed code. In that case the assembler encodes all instructions as they are.

Optimize jump chains (option -Oj/-OJ)

When this option is enabled, the assembler replaces chained jumps by a single jump instruction. For example, a jump from a to b immediately followed by a jump from b to c, is replaced by a jump from a to c. Note that this optimization has no effect on compiled C files, because jump chains are already optimized by the compiler. By default this option is disabled.

Optimize instruction size (option -Os/-OS)

When this option is enabled, the assembler tries to find the shortest possible operand encoding for instructions. By default this option is enabled.

4.5. Generating a List File

The list file is an additional output file that contains information about the generated code. You can customize the amount and form of information.

If the assembler generates errors or warnings, these are reported in the list file just below the source line that caused the error or warning.

Example on the command line (Windows Command Prompt)

The following command generates the list file `test.lst`:

```
asppc -l test.asm
```

See Section 9.1, *Assembler List File Format*, for an explanation of the format of the list file.

4.6. Assembler Error Messages

The assembler reports the following types of error messages.

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 with the assembler option **--no-warnings**.

Display detailed information on diagnostics

On the command line you can use the assembler option **--diag** to see an explanation of a diagnostic message:

```
asppc --diag=[format:]{all | number,...]
```

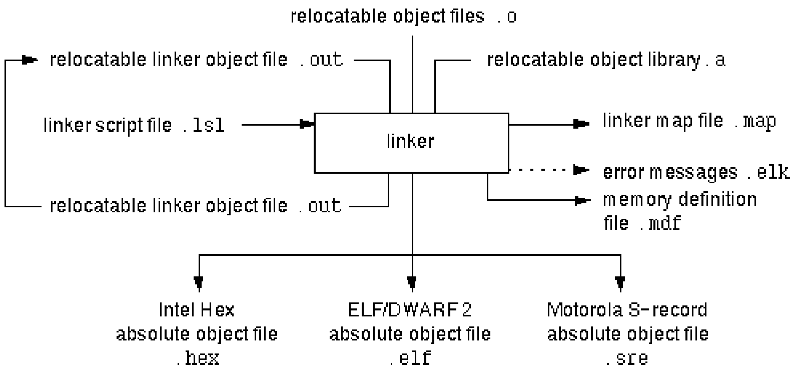
Chapter 5. Using the Linker

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:


relocatable object files .o
relocatable linker object file .out
relocatable object library .a
linker script file .lsl
linker
linker map file .map
error messages .elk
memory definition file .mdf
relocatable linker object file .out
Intel Hex absolute object file .hex
ELF/DWARF 2 absolute object file .elf
Motorola S-record absolute object file .sre

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 7.3, *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.

5.1. 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.

5.1.1. 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.

5.1.2. Phase 2: Locating

In the locating phase, the linker assigns absolute addresses to the object code, placing each section in a specific part of the target memory. The linker also replaces references to symbols by the actual address of those symbols. The resulting file is an absolute object file which you can actually load into a target memory. Optionally, when the resulting file should be loaded into a ROM device the linker creates a so-called copy table section which is used by the startup code to initialize the data and BSS sections.

Code modification

When the linker assigns absolute addresses to the object code, it needs to modify this code according to certain rules or *relocation expressions* to reflect the new addresses. These relocation expressions are stored in the relocatable object file. Consider the following snippet of x86 code that moves the contents of variable a to variable b via the `eax` register:

```
A1 3412 0000 mov a,%eax   (a defined at 0x1234, byte reversed)
A3 0000 0000 mov %eax,b   (b is imported so the instruction refers to
                            0x0000 since its location is unknown)
```

Now assume that the linker links this code so that the section in which a is located is relocated by 0x10000 bytes, and b turns out to be at 0x9A12. The linker modifies the code to be:

```
A1 3412 0100 mov a,%eax   (0x10000 added to the address)
A3 129A 0000 mov %eax,b   (0x9A12 patched in for b)
```

These adjustments affect instructions, but keep in mind that any pointers in the data part of a relocatable object file have to be modified as well.

Output formats

The linker can produce its output in different file formats. The default ELF/DWARF 2 format (`.elf`) contains an image of the executable code and data, and can contain additional debug information. The Intel-Hex format (`.hex`) and Motorola S-record format (`.sre`) only contain an image of the executable code and data. You can specify a format with the options **--output** (**-o**) and **--chip-output** (**-c**).

Controlling the linker

Via a so-called *linker script file* you can gain complete control over the linker. The script language is called the *Linker Script Language* (LSL). Using LSL you can define:

- The memory installed in the embedded target system:

To assign locations to code and data sections, the linker must know what memory devices are actually installed in the embedded target system. For each physical memory device the linker must know its start-address, its size, and whether the memory is read-write accessible (RAM) or read-only accessible (ROM).

- How and where code and data should be placed in the physical memory:

  Embedded systems can have complex memory systems. If for example on-chip and off-chip memory devices are available, the code and data located in internal memory is typically accessed faster and with dissipating less power. To improve the performance of an application, specific code and data sections should be located in on-chip memory. By writing your own LSL file, you gain full control over the locating process.

- The underlying hardware architecture of the target processor.

  To perform its task the linker must have a model of the underlying hardware architecture of the processor you are using. For example the linker must know how to translate an address used within the object file (a logical address) into an offset in a particular memory device (a physical address). In most linkers this model is hard coded in the executable and can not be modified. For the TASKING linker this hardware model is described in the linker script file. This solution is chosen to support configurable cores that are used in system-on-chip designs.

When you want to write your own linker script file, you can use the standard linker script files with architecture descriptions delivered with the product.

See also Section 5.7, *Controlling the Linker with a Script*.

5.2. Calling the Linker

The TASKING VX-toolset for Power Architecture has a make utility to build your entire project and you can use the control program to invoke the tools in the toolset. It also possible to invoke all tools individually.

Invocation syntax on the command line (Windows Command Prompt):

`lkppc` `[ [`*`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 5.3, *Linking with Libraries*.

You can find a detailed description of all linker options in Section 7.3, *Linker Options*.

Example:

```
lkppc -ddefault.lsl test.o
```

This links and locates the file `test.o` and generates the file `test.elf`.

5.3. 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>\lib\e200   (Power Architecture e200x libraries)
```

An overview of the system libraries is given in the following table:

Libraries	Description

libc[v][l][s].a	C libraries
	Optional letter:
	v = VLE instructions (no compiler option **--no-vle**)
	l = little-endian (compiler option **--endianness=little**)
	s = single precision floating-point (compiler option **--no-double**)

libfp[v][l][t].a	Floating-point libraries
	Optional letter:
	v = VLE instructions (no compiler option **--no-vle**)
	l = little-endian (compiler option **--endianness=little**)
	t = trapping (control program option **--fp-trap**)

librt[v][l].a	Run-time library
	Optional letter:
	v = VLE instructions (no compiler option **--no-vle**)
	l = little-endian (compiler option **--endianness=little**)

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.

To link the default C (system) 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.a`, type:

```
lkppc --library=c test.o
```

User library

You can create your own libraries. Section 6.3, *Archiver* describes how you can use the archiver to create your own library with object modules.

To link user libraries

When you want to link user libraries from the command line, you must specify their filenames on the command line:

```
lkppc start.o mylib.a
```

If the library resides in a sub-directory, specify that directory with the library name:

```
lkppc start.o mylibs\mylib.a
```

If you do not specify a directory, the linker searches the library in the current directory only.

Library order

The order in which libraries appear on the command line is important. By default the linker processes object files and libraries in the order in which they appear at the command line. Therefore, when you use a weak symbol construction, like `printf`, in an object file or your own library, you must position this object/library before the C library.

With the option **--first-library-first** you can tell the linker to scan the libraries from left to right, and extract symbols from the first library where the linker finds it. This can be useful when you want to use newer versions of a library routine:

```
lkppc --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`.

5.3.1. 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 directories that are specified with the option **--library-directory (-L)**. If you specify the **-L** option without a pathname, the linker stops searching after this step.

2. When the linker did not find the library (because it is not in the specified library directory or because no directory is specified), it looks in the path(s) specified in the environment variable `LIBPPC`.

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.

5.3.2. How the Linker Extracts Objects from Libraries

A library built with the TASKING archiver **arppc** 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:

```
lkppc 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**):

```
lkppc --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.

5.4. 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.

```
lkppc --incremental test1.o -otest.out
lkppc test2.o test.out
```

This links the file `test1.o` and generates the file `test.out`. This file is used again and linked together with `test2.o` to create the file `test.elf` (the default name if no output filename is given in the default ELF/DWARF 2 format).

With incremental linking it is normal to have unresolved references in the output file until all `.o` files are linked and the final `.out` or `.elf` file has been reached. The option **--incremental** (**-r**) for incremental linking also suppresses warnings and errors because of unresolved symbols.

5.5. Importing Binary Files

With the TASKING linker it is possible to add a binary file to your absolute output file. In an embedded application you usually do not have a file system where you can get your data from. With the linker option **--import-object** you can add raw data to your application. This makes it possible for example to display images on a device or play audio. The linker puts the raw data from the binary file in a section. The section is aligned on a 4-byte boundary. The section name is derived from the filename, in which dots are replaced by an underscore. So, when importing a file called `my.mp3`, a section with the name `my_mp3` is created. In your application you can refer to the created section by using linker labels.

For example:

```
#include <stdio.h>
__data extern char   _lc_ub_my_mp3; /* linker labels */
__data 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]);
}
```

Because the compiler does not know in which space the linker will locate the imported binary, you have to make sure the symbols refer to the same space in which the linker will place the imported binary. You do this by using the memory qualifier `__data`, otherwise the linker cannot bind your linker symbols.

5.6. Linker Optimizations

During the linking and locating phase, the linker looks for opportunities to optimize the object code. Both code size and execution speed can be optimized.

You can enable or disable the optimizations described below. The command line option for each optimization is given in brackets.

Delete unreferenced sections (option -Oc/-OC)

This optimization removes unused sections from the resulting object file.

Optimize loads (for speed) while linking (option -Og/-OG)

This optimization performs global linker optimizations on code section contents. It replaces instructions that do a direct read from ROM by instructions using the value in ROM as an immediate operand. For example:

```
lwz r3,@relsda(a)(r2)
```

will be replaced by

```
li r3,5
```

when the `@relsda(a)(r2)` points to a ROM location with a value 5.

In general this is a speed optimization because it avoids memory reads.

First fit decreasing (option -Ol/-OL)

When the physical memory is fragmented or when address spaces are nested it may be possible that a given application cannot be located although the size of the available physical memory is larger than the sum of the section sizes. Enable the first-fit-decreasing optimization when this occurs and re-link your application.

The linker's default behavior is to place sections in the order that is specified in the LSL file (that is, working from low to high memory addresses or vice versa). This also applies to sections within an unrestricted group. If a memory range is partially filled and a section must be located that is larger than the remainder of this range, then the section and all subsequent sections are placed in a next memory range. As a result of this gaps occur at the end of a memory range.

When the first-fit-decreasing optimization is enabled the linker will first place the largest sections in the smallest memory ranges that can contain the section. Small sections are located last and can likely fit in the remaining gaps.

Compress copy table (option -Ot/-OT)

The startup code initializes the application's data areas. The information about which memory addresses should be zeroed and which memory ranges should be copied from ROM to RAM is stored in the copy table.

When this optimization is enabled the linker will try to locate sections in such a way that the copy table is as small as possible thereby reducing the application's ROM image.

Delete duplicate code (option -Ox/-OX)

This optimization removes code that is defined more than once, from the resulting object file.

5.7. Controlling the Linker with a Script

With the options on the command line you can control the linker's behavior to a certain degree. If you want more control over the locating process you can supply a script to the linker, in which you can specify where your sections will be located, how much memory is available, which sorts of memory are available, and so on.

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.

5.7.1. Purpose of the Linker Script Language

The Linker Script Language (LSL) serves three purposes:

1. It provides the linker with a definition of the target's core architecture. This definition is supplied with the toolset.

2. It provides the linker with a specification of the memory attached to the target processor.

3. It provides the linker with information on how your application should be located in memory. This gives you, for example, the possibility to create overlaying sections.

The linker accepts multiple LSL files. You can use the specifications of the core architectures that Altium has supplied in the `include.lsl` directory. Do not change these files.

If you use a different memory layout than described in the LSL file supplied for the target core, you must specify this in a separate LSL file and pass both the LSL file that describes the core architecture and your LSL file that contains the memory specification to the linker. Next you may want to specify how sections should be located and overlaid. You can do this in the same file or in another LSL file.

LSL has its own syntax. In addition, you can use the standard C preprocessor keywords, such as `#include` and `#define`, because the linker sends the script file first to the C preprocessor before it starts interpreting the script.

The complete LSL syntax is described in Linker Script Language.

5.7.2. Structure of a Linker Script File

A script file consists of several definitions. The definitions can appear in any order.

The architecture definition (required)

In essence an *architecture definition* describes how the linker should convert logical addresses into physical addresses for a given type of core. If the core supports multiple address spaces, then for each space the linker must know how to perform this conversion. In this context a physical address is an offset on a given internal or external bus. Additionally the architecture definition contains information about items such as the (hardware) stack and the interrupt vector table.

This specification is normally written by Altium. Altium supplies LSL files in the `include.lsl` directory. The file `ppc_arch.lsl` defines the base architecture for all cores. The files `default.lsl` and `e200.lsl` extend the base architecture for each Power Architecture core.

The architecture definition of the LSL file should not be changed by you unless you also modify the core's hardware architecture. If the LSL file describes a multi-core system an architecture definition must be available for each different type of core.

The linker uses the architecture name in the LSL file to identify the target. For example, the default library search path can be different for each core architecture.

The derivative definition

The *derivative definition* describes the configuration of the internal (on-chip) bus and memory system. Basically it tells the linker how to convert offsets on the buses specified in the architecture definition into offsets in internal memory. Microcontrollers and DSPs often have internal memory and I/O sub-systems apart from one or more cores. The design of such a chip is called a *derivative*.

When you want to use multiple cores of the same type, you must instantiate the cores in a derivative definition, since the linker automatically instantiates only a single core for an unused architecture.

Altium supplies LSL files for each derivative (*`derivative`*`.lsl`). 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 PPC architecture, its external memory and how sections are located in memory, may have the following skeleton:

```
architecture PPC
{
    // Specification of the PPC core architecture.
    // Written by Altium.
}

derivative default_derivative  // derivative name is arbitrary
{
    // Specification of the derivative.
    // Written by Altium.
    core ppc         // always specify the core
    {
       architecture = PPC;
    }

    bus local_bus    // internal bus
    {
       // maps to bus "local_bus" in "ppc" core
    }

    // internal memory
}

processor spe        // processor name is arbitrary
{
    derivative = default_derivative;

    // You can omit this part, except if you use a
    // multi-core system.
}

memory ext_name
{
    // external memory definition
}

section_layout spe:ppc:linear    // section layout
```

```
{
    // section placement statements

    // sections are located in address space 'linear'
    // of core ppc of processor 'spe'
}
```

Overview of LSL files delivered by Altium

Altium supplies the following LSL files in the directory `include.lsl`.

LSL file	**Description**
`ppc_arch.lsl`	Defines the base architecture (PPC) for all cores. Contains a section layout.
`*derivative*.lsl`	Defines the derivative and defines a single processor. Contains a memory definition. It includes the file `ppc_arch.lsl`. The selection of the derivative is based on your CPU/core selection (control program option **--core**).
`default.lsl`	Defines a default derivative with memory and defines a single processor. It includes the file `ppc_arch.lsl`. The file `default.lsl` is used on a command line invocation of the tools, when no core is selected (no option **--core**).

On the command line, the linker uses the file `default.lsl`, unless you specify another file with the linker option **--lsl-file** (**-d**).

5.7.3. 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, and relative addressing.

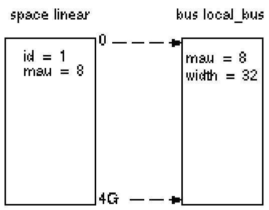
- the addresses in the two address spaces represent different locations so they do not overlap. This means the core must have separate sets of address lines for the address spaces. For example, in Harvard architectures we can identify at least a code and a data memory space.

Address spaces (even nested) can have different minimal addressable units (MAU), alignment restrictions, and page sizes. All address spaces have a number that identifies the logical space (id). The following table lists the different address spaces for the architecture PPC as defined in `ppc_arch.lsl`.

Space	Id	MAU	Description
linear	1	8	Linear address space.

The 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 architecture PPC:



The figure shows one address spaces 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 these address spaces 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 PPC
{
    // All code above goes here.
}
```

5.7.4. 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 ppc
{
    architecture = PPC;
}
```

Bus

Each derivative can contain a bus definition for connecting external memory. In this example, the bus `local_bus` maps to the bus `local_bus` defined in the architecture definition of core `ppc`:

```
bus local_bus
{
    mau = 8;
    width = 32;
    map (dest=bus:ppc:local_bus, dest_offset=0, size=4G);
}
```

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:ppc: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 default_derivative        // name of derivative
{
    // All code above goes here
}
```

5.7.5. 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.

Altium defines a “single processor environment” (spe) in each *derivative*`.lsl` file. For example:

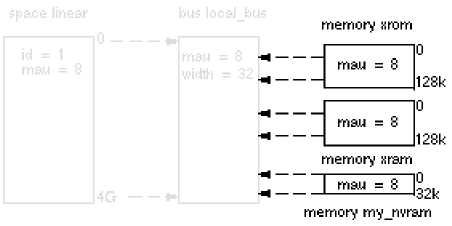
```
processor spe
{
    derivative = default_derivative;
}
```

5.7.6. 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 128kB of external ROM, named `xrom`, 128kB of external RAM, named `xram` 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 xrom
{
    mau = 8;
    type = rom;
    size = 0x20000;
    map ( size = 0x20000, dest_offset=0x00100000, dest=bus:spe:local_bus);
}

memory xram
{
    mau = 8;
    type = ram;
    size = 0x20000;
    map ( size = 0x20000, dest_offset=0x00120000, dest=bus:spe:local_bus);
}

memory my_nvram
{
    mau = 8;
    type = ram;
    size = 32k;
    map ( size = 32k, dest_offset=0x00140000, dest=bus:spe: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 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.

5.7.7. 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
__data int  battery_backup_tag;
__data 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.

By default the compiler creates the section `.sbss` to store uninitialized data objects. With the `__data` qualifier this is `.bss`. This section name tells the linker to locate the section in address space `linear` and that the section content should be filled with zeros at startup.

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_nvsram` from the example in Section 5.7.6, *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
}
```

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_nvsram`. Furthermore, the section should not be cleared and therefore the attribute **s** (scratch) is assigned to the group:

```
group ( ordered, run_addr = mem:my_nvsram, 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 11, *Linker Script Language (LSL)*.

5.8. Linker Labels

The linker creates labels that you can use to refer to from within the application software. Some of these labels are real labels at the beginning or the end of a section. Other labels have a second function, these labels are used to address generated data in the locating phase. The data is only generated if the label is used.

Linker labels are labels starting with `_lc_`. The linker assigns addresses to the following labels when they are referenced:

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.
`_lc_ue_`*name* `_lc_e_`*name*	End of section *name*. Also used to mark the end of the stack or heap.
`_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.

Label	Description
`_lc_ge_`*`name`*	End of group *name*. 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 the C source file `foo.c` contains the following:

```
int myfunc(int a)
{
    /* some source lines */
    return 1;
}
```

This results in a section with the name `.text_vle.myfunc`.

In the following source file `main.c` all dots of the section name are replaced by underscores:

```
#include <stdio.h>
extern void * _lc_ub__text_vle_myfunc;

void main(void)
{
    printf("The function myfunc is located at %p\n",
            &_lc_ub__text_vle_myfunc);
}
```

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 stack
```

5.9. Generating a Map File

The map file is an additional output file that contains information about the location of sections and symbols. You can customize the type of information that should be included in the map file.

Example on the command line (Windows Command Prompt)

The following command generates the map file `test.map`:

```
lkppc --map-file test.o
```

With this command the map file `test.map` is created.

See Section 9.2, *Linker Map File Format*, for an explanation of the format of the map file.

5.10. Linker Error Messages

The linker reports the following types of error messages.

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 with the 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

On the command line you can use the linker option **--diag** to see an explanation of a diagnostic message:

```
lkppc --diag=[format:]{all | number,...]
```

Chapter 6. Using the Utilities

The TASKING VX-toolset for Power Architecture comes with a number of utilities:

ccppc	A control program. The control program invokes all tools in the toolset and lets you quickly generate an absolute object file from C and/or assembly source input files.
amk	A make utility which supports parallelism and utilizes the multiple cores found on modern host hardware.
arppc	An archiver. With this utility you create and maintain library files with relocatable object modules (`.o`) generated by the assembler.
hldumpppc	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.

6.1. 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 sources without the need to invoke the compiler, assembler and linker manually.

The invocation syntax is:

```
ccppc [ [option]... [file]... ]...
```

Recognized input files

- 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-*** (**-Wc**, **-Wa**, **-Wl**) to pass arguments directly to tools.

For a complete list and description of all control program options, see Section 7.4, *Control Program Options*.

Example with verbose output

```
ccppc --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\cppc" --core=e200z0 --no-double -o cc3248a.src test.c
+ "path\asppc" --core=e200z0 -o cc3248b.o cc3248a.src
+ "path\lkppc" -o test.elf --de200z0.lsl --map-file
     cc3248b.o -lcvs -lfpv -lrtv "-Lpath\lib\e200"
```

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

```
ccppc --pass-compiler=-Oc test.c
```

The option **-Oc** is directly passed to the compiler.

6.2. 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".

6.2.1. 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 :
  ccppc test.c   # leading white space
```

Command rules may contain dependencies too. Combining the above for example yields:

```
test.o : test.c
  ccppc 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
    ccppc -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
    ccppc -c test.c -o test.o
    echo the stem is test

filter.o: filter.c
    ccppc -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
`.DEFAULT`	If you call the make utility with a target that has no definition in the makefile, this target is built.

Target	Description
`.DONE`	When the make utility has finished building the specified targets, it continues with the rules following this target.
`.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`.

6.2.2. 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.

6.2.3. Macro Definitions

A *macro* is a symbol name that is replaced with its definition before the makefile is executed. Although the macro name can consist of lower case or upper case characters, upper case is an accepted convention. 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.

6.2.4. 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
```

6.2.5. 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
```

6.2.6. 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).

6.2.7. 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.

6.2.8. 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 7.5, *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.

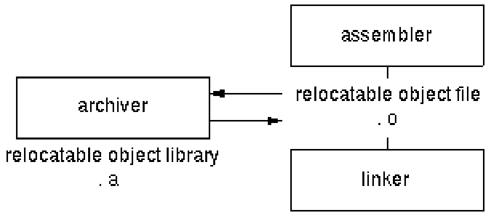
6.3. Archiver

The archiver **arppc** 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.

6.3.1. Calling the Archiver

You can call the archiver from the command line. The invocation syntax is:

```
arppc 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 -u -v**
Extract an object module from the library	**-x**	**-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*	
Create library without notification if library does not exis	**-c**	
Preserve last-modified date from the library	**-o**	
Print symbols in library modules	**-s**{**0**|**1**}	
Replace only newer modules	**-u**	
Verbose	**-v**	
Miscellaneous		
Display options	**-?**	
Display version header	**-V**	
Read options from *file*	**-f** *file*	
Suppress warnings above level *n*	**-w***n*	

For a complete list and description of all archiver options, see Section 7.6, *Archiver Options*.

6.3.2. 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:

```
arppc -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.)

```
arppc -r mylib.a mod3.o
```

Print a list of object modules in the library

To inspect the contents of the library:

```
arppc -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`:

```
arppc -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`:

```
arppc -d mylib.a cstart.o
```

Extract all modules from the library

Extract all modules from the library `mylib.a`:

```
arppc -x mylib.a
```

6.4. HLL Object Dumper

The high level language (HLL) dumper **hldumpppc** 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.

6.4.1. Invocation

Command line invocation

You can call the HLL dumper from the command line. The invocation syntax is:

```
hldumpppc [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 7.7, *HLL Object Dumper Options*. With `hldumpppc --help` you will see the options on `stdout`.

6.4.2. HLL Dump Output Format

The HLL dumper produces output in text format by default, but you can also specify the XML output format with **--output-file-type=xml**. The output is printed on `stdout`, unless you specify an output file with **--output=***filename*.

The parts of the output are dumped in the following order:

1. Module list
2. Section list
3. Section dump (disassembly)
4. HLL symbol table
5. Assembly level symbol table

With the option **--dump-format=***flag* you can control which parts are shown. By default, all parts are shown.

Example

Suppose we have a simple "Hello World" program in a file called `hello.c`. We call the control program as follows:

```
ccppc -g -t hello.c
```

Option **-g** tells to include DWARF debug information. Option **-t** tells to keep the intermediate files. This command results (among other files) in the file `hello.elf` (the absolute output file).

We can dump information about the object file with the following command:

```
hldumpppc hello.elf

---------- Module list ----------

Name    Full path
hello.c hello.c

---------- Section list ----------

Address  Size   Align Name
000000e0      4     4 [.sdata.hello.world]
0000050c     16     4 .text_vle.hello.main
000005c0      6     1 .rodata.hello.$1$str
000005c6     12     1 .rodata.hello.$2$str
400000cc      4     4 .sdata.hello.world

---------- Section dump ----------

    .section [.sdata.hello.world]
    .org 000000e0
    .db 00,00,05,c0                                    ; ....
    .endsec

                                   .section .text_vle.hello.main
0000050c 7060e000 main             e_lis          r3,0
00000510 1c6305c6                  e_add16i       r3,r3,0x5c6
00000514 508d80cc                  e_lwz          r4,-0x7f34(r13)
00000518 78000004                  e_b            0x51c
                                   .endsec

    .section .rodata.hello.$1$str
    .org 000005c0
    .db 77,6f,72,6c,64,00                              ; world.
    .endsec

    .section .rodata.hello.$2$str
    .org 000005c6
    .db 48,65,6c,6c,6f,2c,20,25,73,21,0a,00            ; Hello, %s!..
    .endsec

---------- HLL symbol table ----------

Address    Size HLL Type             Name
0000050c     16 void                 main()
```

```
400000cc      4 unsigned char      * world [hello.c]
400000d0     20 struct               _dbg_request [dbg.c]
400000e5     80 static unsigned char stdin_buf[80] [_iob.c]
40000135     80 static unsigned char stdout_buf[80] [_iob.c]

---------- assembly level symbol table ----------

Address  Size    Type Name
00000000
00000000              hello.src
000000e8      54 code _START
0000050c      16 code main
0000051c      98 code printf
400000cc       4 data world
```

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.
Name	The name of the section.

With option **--sections=***name*[,*name*]... you can specify a list of sections that should be dumped.

Section dump

This part contains the disassembly. It consists of the following columns:

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]**. With option **--data-dump-format=directives** (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.

With option **--data-dump-format=hex**, no directives will be generated for data sections, but data sections are dumped as hexadecimal code with ASCII translation. This only applies to ROM sections. The hex dump has 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 33 (.rodata.hello.$2$str):
00000000 48 65 6c 6c 6f 2c 20 25 73 21 0a 00             Hello, %s!..
```

With option **--data-dump-format=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:

```
400000e5     80 static unsigned char stdin_buf[80] [_iob.c]
```

HLL struct and union symbols are listed by default without fields. For example:

```
400000d0     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:

```
400000d0     20 struct               _dbg_request [dbg.c]
400000d0      4   int                  _errno
400000d4      1   unsigned char        nr
400000d8     12   union                u
400000d8      4     struct               exit
400000d8      4       int                  status
400000d8      8     struct               open
400000d8      4       const unsigned char * pathname
400000dc      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:

```
0000051c     98 int                  printf(const unsigned char * format, ...)
```

The local and static symbols get an identification between square brackets. The filename is printed if 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.

Chapter 7. Tool Options

This chapter provides a detailed description of the options for the compiler, assembler, linker, control program, make utility, archiver and the HLL object dumper.

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:

```
cppc -Oac test.c
cppc --optimize=+coalesce,+cse test.c
```

When you do not specify an option, a default value may become active.

7.1. C Compiler Options

This section lists all C compiler options.

C compiler option: --build-runtime

Command line syntax

```
--build-runtime
```

Description

Special option for building the C run-time library. Used to indicate that the 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 library.

Related information

-

C compiler option: --cert

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 12, *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:

```
cppc --cert=str30 test.c
```

Related information

Chapter 12, *CERT C Secure Coding Standard*

C compiler option **--diag** (Explanation of diagnostic messages)

C compiler option: --check

Command line syntax

```
--check
```

Description

With this option you can check the source code for syntax errors, without generating code. This saves time in developing your application because the code will not actually be compiled.

The compiler reports any warnings and/or errors.

Related information

Assembler option **--check** (Check syntax)

C compiler option: --code-section-alignment

Command line syntax

`--code-section-alignment=``*alignment*`

Default: **0**

Description

By default a code section is aligned to the minimum alignment required by the function that is allocated in the section. With this option you can increase the alignment. The *alignment* (in bytes) must be 0 or a power of two. An alignment of 0 aligns functions to their minimum alignment.

This option does not apply to functions that:

- require a larger alignment than specified by this option
- are explicitly aligned using: `__attribute__((align(`*alignment*`)))`
- are located at an absolute address using: `__attribute__((at(`*address*`)))`

Related information

C compiler option **--data-section-alignment** (Increase alignment of data objects)

C compiler option: --compact-max-size

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:

```
cppc --optimize=+compact --compact-max-size=100 test.c
```

Related information

C compiler option **--optimize=+compact** (Optimization: code compaction)

C compiler option **--max-call-depth** (Maximum call depth for code compaction)

C compiler option: --constant-data-memory

Command line syntax

`--constant-data-memory=`*`space`*

You can specify the following *space* arguments:

> **__data**
>
> **__sdata**
>
> **__sdata2**
>
> **__sdata0**
>
> **threshold**

Default: **threshold**

Description

With this option you can control the allocation of constant data objects. Constant data objects are `const` variables and automatic initializers. Constant data objects that are not explicitly qualified, are allocated in the *space* specified by this option.

The default is **threshold**, which means that constant data objects are allocated according the **--default-sdata*-size** threshold options.

Related information

Pragma `constant_data_memory`

C compiler option **--data-memory** (Assign memory to non-constant data objects)

C compiler option **--string-literal-memory** (Assign memory to string literals)

C compiler option: --core

Command line syntax

--core=*core*

You can specify the following *core* arguments:

e200z0	e200z0 core
e200z3	e200z3 core
e200z4	e200z4 core
e200z6	e200z6 core
e200z7	e200z7 core

Default: **e200z0**

Description

With this option you specify the core architecture for the target processor for which you create your application.

The macro `__CORE_E200Z0__` is defined in the C source file.

Example

Specify a core:

```
cppc --core=e200z6 test.c
```

Related information

-

C compiler option: --data-memory

Command line syntax

`--data-memory=`*`space`*

You can specify the following *space* arguments:

- **__data**
- **__sdata**
- **__sdata2**
- **__sdata0**
- **threshold**

Default: **threshold**

Description

With this option you can control the allocation of non-constant data objects. Data objects that are not explicitly qualified, are allocated in the *space* specified by this option.

The default is **threshold**, which means that data objects are allocated according the **--default-sdata*-size** threshold options.

Related information

Pragma `data_memory`

C compiler option **--constant-data-memory** (Assign memory to constant data objects)

C compiler option **--string-literal-memory** (Assign memory to string literals)

C compiler option: --data-section-alignment

Command line syntax

`--data-section-alignment=`*`alignment`*

Default: **0**

Description

By default a data section is aligned to the minimum alignment required by the object that is allocated in the section. With this option you can increase the alignment. The *alignment* (in bytes) must be 0 or a power of two. An alignment of 0 aligns data objects to their minimum alignment.

This option does not apply to:

- objects that require a larger alignment than specified by this option
- objects that are explicitly aligned using: `__attribute__((align(`*alignment*`)))`
- objects that are located at an absolute address using: `__attribute__((at(`*address*`)))`
- automatic objects
- struct members

This option does apply to:

- global variables
- string literals
- static initializers
- switch tables

Related information

C compiler option **--code-section-alignment** (Increase alignment of functions)

C compiler option: --debug-info (-g)

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: **`--debug-info`** (same as **`--debug-info=default`**)

Description

With this option you tell the compiler to add directives to the output file for including symbolic information. This facilitates high level debugging but increases the size of the resulting assembler file (and thus the size of the object file). For the final application, compile your C files without debug information.

The DWARF debug format allows for a flexible approach as to how much symbolic information is included, as long as the structure is valid. Adding all possible DWARF data for a program is not practical. The amount of DWARF information per compilation unit can be huge. And for large projects, with many object modules the link time can grow unacceptably long. That is why the compiler has several debug information levels. In general terms one can say, the higher the level the more DWARF information is produced.

The DWARF data in an object module is not only used for debugging. The toolset can also do "type checking" of the whole application. In that case the linker will use the DWARF information of all object modules to determine if every use of a symbol is done with the same type. In other words, if the application is built with type checking enabled then the compiler will add DWARF information too.

Small set of debug information

With this suboption only DWARF call frame information and type information are generated. This enables you to inspect parameters of nested functions. The type information improves debugging. You can perform a stack trace, but stepping is not possible because debug information on function bodies is not generated. You can use this suboption, for example, to compact libraries.

Default debug information

This provides all debug information you need to debug your application. It meets the debugging requirements in most cases without resulting in oversized assembler/object files.

Full debug information

With this suboption extra debug information is generated about unused typedefs and DWARF "lookup table sections". Under normal circumstances this extra debug information is not needed to debug the program. Information about unused typedefs concerns all typedefs, even the ones that are not used for any variable in the program. (Possibly, these unused typedefs are listed in the standard include files.) With this suboption, the resulting assembler/object file will increase significantly.

In the following table you see in more detail what DWARF information is included for the debug option levels.

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: --default-sdata-size

Command line syntax

`--default-sdata-size=``[`*`min`*`,]`*`max`*

Default: **`--default-sdata-size=0,8`**

Description

By default, all data is allocated in the `__data` memory space. With this option you can specify that unqualified objects with a size greater than *min* and smaller than or equal to *max* bytes can be allocated in the `__sdata` memory space automatically.

Option **--default-sdata-type** controls the type of objects that are affected (constant or non-constant).

The values must be specified in bytes. When *min* is omitted, 0 is assumed. Objects that are absolute are not moved.

Example

To put all unqualified non-constant data objects with a size of 12 bytes or smaller in `__sdata`, enter:

```
cppc --default-sdata-size=12 --default-sdata-type=non-const --test.c
```

Related information

C compiler option **--default-sdata-type** (object type for object to be placed in `__sdata`)

Section 1.2.1, *Memory Qualifiers*

C compiler option: --default-sdata-type

Command line syntax

`--default-sdata-type=`*`flag`*`,...`

You can set the following flags:

+/-const	**c/C**	select const objects
+/-non-const	**n/N**	select non-const objects

Default: **`--default-sdata-type=Cn`**

Description

With this option you can select the type of objects that are affected by option **--default-sdata-size**.

Note that the generated code must conform to the EABI. This means that when constant variables are allocated in `__sdata` memory space, they are allocated in an initialized data section.

Example

To put all unqualified constant and non-constant data objects with a size of 12 bytes or smaller in `__sdata`, enter:

```
cppc --default-sdata-size=12 --default-sdata-type=cn --test.c
```

Related information

Pragma `default_sdata_type`

C compiler option **--default-sdata-size** (size in bytes for data elements that are by default located in `__sdata` sections)

Section 1.2.1, *Memory Qualifiers*

C compiler option: --default-sdata0-size

Command line syntax

`--default-sdata0-size=``[`*`min`*`,]`*`max`*

Default: **`--default-sdata0-size=0,0`**

Description

By default, all data is allocated in the `__data` memory space. With this option you can specify that unqualified objects with a size greater than *min* and smaller than or equal to *max* bytes can be allocated in the `__sdata0` memory space automatically.

Option **--default-sdata0-type** controls the type of objects that are affected (constant or non-constant).

The values must be specified in bytes. When *min* is omitted, 0 is assumed. Objects that are absolute are not moved.

Example

To put all unqualified constant data objects with a size of 8 bytes or smaller in `__sdata0`, enter:

```
cppc --default-sdata0-size=8 --default-sdata0-type=cN --test.c
```

Related information

C compiler option **--default-sdata0-type** (object type for object to be placed in `__sdata0`)

Section 1.2.1, *Memory Qualifiers*

C compiler option: --default-sdata0-type

Command line syntax

`--default-sdata0-type=``*flag*,...`

You can set the following flags:

+/-const	**c/C**	select const objects
+/-non-const	**n/N**	select non-const objects

Default: **`--default-sdata0-type=cN`**

Description

With this option you can select the type of objects that are affected by option **--default-sdata0-size**.

Note that the generated code must conform to the EABI. This means that when constant objects are mixed with non-constant objects in the `__sdata0` memory space, all romdata sections will be converted to initialized data sections (by the linker). This costs twice the amount of memory (a ROM copy for each variable), and the startup code must copy ROM to RAM, which costs time.

Note also that on some targets only ROM memory is available in `__sdata0` space. This may cause locate errors when you try to allocate non-constant objects in this space.

Example

To put all unqualified constant data objects with a size of 8 bytes or smaller in `__sdata0`, enter:

```
cppc --default-sdata0-size=8 --default-sdata0-type=cN --test.c
```

Related information

Pragma `default_sdata0_type`

C compiler option **--default-sdata0-size** (size in bytes for data elements that are by default located in `__sdata0` sections)

Section 1.2.1, *Memory Qualifiers*

C compiler option: --default-sdata2-size

Command line syntax

`--default-sdata2-size=``[`*`min`*`,]`*`max`*

Default: **`--default-sdata2-size=0,8`**

Description

By default, all data is allocated in the `__data` memory space. With this option you can specify that unqualified objects with a size greater than *min* and smaller than or equal to *max* bytes can be allocated in the `__sdata2` memory space automatically.

Option **--default-sdata2-type** controls the type of objects that are affected (constant or non-constant).

The values must be specified in bytes. When *min* is omitted, 0 is assumed. Objects that are absolute are not moved.

Example

To put all unqualified constant data objects with a size of 8 bytes or smaller in `__sdata2`, enter:

```
cppc --default-sdata2-size=8 --default-sdata2-type=cN --test.c
```

Related information

C compiler option **--default-sdata2-type** (object type for object to be placed in `__sdata2`)

Section 1.2.1, *Memory Qualifiers*

C compiler option: --default-sdata2-type

Command line syntax

`--default-sdata2-type=`*`flag`*`,...`

You can set the following flags:

+/-const	**c/C**	select const objects
+/-non-const	**n/N**	select non-const objects

Default: **`--default-sdata2-type=cN`**

Description

With this option you can select the type of objects that are affected by option **--default-sdata2-size**.

Note that the generated code must conform to the EABI. This means that when constant objects are mixed with non-constant objects in the `__sdata2` memory space, all romdata sections will be converted to initialized data sections (by the linker). This costs twice the amount of memory (a ROM copy for each variable), and the startup code must copy ROM to RAM, which costs time.

Example

To put all unqualified constant data objects with a size of 8 bytes or smaller in `__sdata2`, enter:

```
cppc --default-sdata2-size=8 --default-sdata2-type=cN --test.c
```

Related information

Pragma `default_sdata2_type`

C compiler option **--default-sdata2-size** (size in bytes for data elements that are by default located in `__sdata2` sections)

Section 1.2.1, *Memory Qualifiers*

C compiler option: --define (-D)

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, 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 compiler with the option **--option-file** (**-f**) *file*.

Defining macros with this option (instead of in the C source) is, for example, useful to compile conditional C source as shown in the example below.

Example

Consider the following C program with conditional code to compile a demo program and a real program:

```
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:

```
cppc --define=DEMO test.c
cppc --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.

```
cppc --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

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

```
cppc --dep-file=test.dep test.c
```

The compiler compiles the file `test.c`, which results in the output file `test.src`, and generates dependency lines in the file `test.dep`.

Related information

C compiler option **--preprocess=+make** (Generate dependencies for make)

C compiler option: --diag

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:

```
cppc --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:

```
cppc --diag=html:all > cerrors.html
```

Related information

Section 3.9, *C Compiler Error Messages*

C compiler option **--cert** (Enable individual CERT checks)

C compiler option: --dwarf-version

Command line syntax

```
--dwarf-version={2|3}
```

Default: **3** (or **2** when **--eabi=D** is selected)

Description

With this option you tell the compiler which DWARF debug version to generate, DWARF2 or DWARF3 (default). When the option **--eabi=D** is set, only DWARF version 2 is allowed, and is automatically selected when **--dwarf-version** is not used.

Related information

Section 10.1, *ELF/DWARF Object Format*

C compiler option: --eabi

Command line syntax

`--eabi=`*`flags`*

You can set the following flags:

+/-dwarf	**d/D**	allow an alternative DWARF version
+/-extend	**e/E**	sign/zero extend small integrals
+/-long-double	**l/L**	allow long double type
+/-no-clear	**n/N**	allow the use of option **--no-clear**
+/-sdata0-rom	**r/R**	allow romdata in sdata0 space
+/-volatile-sdata	**v/V**	move volatile objects to short addressable space

Default: **`--eabi=dElnrv`**

Description

With this option you control the level of EABI compliancy.

With **--eabi=+dwarf**, another DWARF version than the prescribed 2.0.0 is allowed.

With **--eabi=+extend**, small integral types (`short` and `char`) will be sign/zero extended when they are passed to or returned from functions.

Note that the libraries delivered with the product have been built with **--eabi=-extend**. So, if you change this option you also have to rebuild the libraries.

With **--eabi=+long-double**, `long double` types are accepted, and will be treated as `double`.

With **--eabi=+no-clear**, the use of option **--no-clear** and `#pragma noclear` is allowed.

With **--eabi=+sdata0-rom**, `const` variables in the `__sdata0` space are placed in a romdata section. Without this option an initialized data section will be used.

With **--eabi=+volatile-sdata**, volatile objects can be moved to a short addressable space under control of the options **--default-sdata-size**, **--default-sdata2-size** and **--default-sdata0-size**.

Related information

C compiler option **--eabi-compliant** (code needs to be completely EABI compliant)

C compiler option **--default-sdata-size** (size in bytes for data elements that are by default located in `__sdata` sections)

C compiler option **--default-sdata2-size** (size in bytes for data elements that are by default located in `__sdata2` sections)

C compiler option **--default-sdata0-size** (size in bytes for data elements that are by default located in `__sdata0` sections)

C compiler option **--no-clear** (do not clear non-initialized global/static variables)

C compiler option: --eabi-compliant

Command line syntax

```
--eabi-compliant
```

Description

Use this option when the generated code needs to be completely EABI compliant.

This option is an alias for **--eabi=DeLNRV**.

Related information

C compiler option **--eabi** (control level of EABI compliancy)

C compiler option: --endianness

Command line syntax

`--endianness=`*`endianness`*

`--little-endian`

You can specify the following *endianness*:

big	**b**	Big endian (default)
little	**l**	Little endian

Description

By default, the compiler generates code for a big-endian target (most significant byte of a word at lowest byte address). With **--endianness=little** the compiler generates code for a little-endian target (least significant byte of a word at lowest byte address). **--little-endian** is an alias for option **--endianness=little**.

The endianness used must be a valid one for the architecture you are compiling for.

Related information

-

C compiler option: --error-file

Command line syntax

`--error-file``[=`*`file`*`]`

Description

With this option the compiler redirects error messages to a file. If you do not specify a filename, the error file will be named after the input file with extension `.err`.

Example

To write errors to `errors.err` instead of `stderr`, enter:

```
cppc --error-file=errors.err test.c
```

Related information

-

C compiler option: --fp-trap

Command line syntax

```
--fp-trap
```

Description

With this option you tell the compiler to allow trapping of floating-point exceptions.

The floating-point instructions, as implemented in the FPU, need to be handled in a special way if floating-point trapping behavior is expected from the generated code. A trapped floating-point library is required. The compiler may use slightly different instructions in some minor cases, to make sure that the hardware will generate an exception when necessary. In addition, instruction scheduling will be slightly adjusted in order to achieve optimal performance.

Related information

Control program option **--fp-trap** (Use trapped floating-point library)

Section 5.3, *Linking with Libraries*

C compiler option: --fpu-version

Command line syntax

`--fpu-version=`*`version`*

You can specify the following arguments:

auto	Automatic. The C compiler chooses the floating-point unit based upon the selected core.
efpu-v1.1	Embedded floating-point unit (EFPU) v1.1
efpu-v2.0	Embedded floating-point unit (EFPU) v2.0

Description

With this option you can select the version of the floating-point unit. You can only select versions lower than the version of the default FPU.

Related information

C compiler option **--no-fpu** (Do not use FPU)

C compiler option: --help (-?)

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:

```
cppc -?
cppc --help
cppc
```

The following invocation displays a list of the available pragmas:

```
cppc --help=pragmas
```

Related information

-

C compiler option: --include-directory (-I)

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 `CPPCINC` 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:

```
cppc --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)

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

```
cppc --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

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:

```
cppc --optimize=+compact --inline test.c
```

Related information

C compiler option **--optimize=+compact** (Optimization: code compaction)

Section 1.8.3, *Inlining Functions: inline*

C compiler option: --inline-max-incr / --inline-max-size

Command line syntax

```
--inline-max-incr=percentage  (default: -1)
--inline-max-size=threshold   (default: -1)
```

Description

With these options you can control the automatic function inlining optimization process of the compiler. These options have only effect when you have enabled the inlining optimization (option **--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

```
cppc --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 1.8.3, *Inlining Functions: inline*

Section 3.6.3, *Optimize for Size or Speed*

C compiler option: --iso (-c)

Command line syntax

`--iso=``{`**`90`**`|`**`99`**`}`

`-c``{`**`90`**`|`**`99`**`}`

Default: **`--iso=99`**

Description

With this option you select the ISO C standard. C90 is also referred to as the "ANSI C standard". C99 refers to the newer ISO/IEC 9899:1999 (E) standard. C99 is the default.

Example

To select the ISO C90 standard on the command line:

```
cppc --iso=90 test.c
```

Related information

C compiler option **--language** (Language extensions)

C compiler option: --keep-output-files (-k)

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

```
cppc --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)

Command line syntax

`--language=[`*`flags`*`]`

`-A[`*`flags`*`]`

You can set the following flags:

+/-gcc	**g/G**	enable a number of gcc extensions
+/-comments	**p/P**	// comments in ISO C90 mode
+/-volatile	**v/V**	don't optimize across volatile access
+/-strings	**x/X**	relaxed const check for string literals

Default: **`-AGpVx`**

Default (without flags): **`-AGPVX`**

Description

With this option you control the language extensions the compiler can accept.

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.
- An `__attribute__` directly following a struct/union definition relates to that tag instead of to the objects in the declaration.

For a more complete description of these extensions, you can refer to the UNIX gcc info pages (**info gcc**).

Comments in ISO C90 mode

With **--language=+comments** (**-Ap**) you tell the compiler to allow C++ style comments (//) in ISO C90 mode (option **--iso=90**). In ISO C99 mode this style of comments is always accepted.

Check assignment 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;
void main( void ) { p = "hello"; }
```

Example

```
cppc --language=-comments,+strings --iso=90 test.c
cppc -APx -c90 test.c
```

The compiler compiles in ISO C90 mode, accepts assignments of a constant string to a non-constant string pointer and does not allow C++ style comments.

Optimization across volatile access

With the **--language=+volatile** (**-Av**) option, the compiler will block optimizations when reading or writing a volatile object, by treating the access as a call to an unknown function. 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
    lwz    r0,@relsda(variable)(r13)  ; <== Moved across volatile access
    li     r7,0
    stw    r7,@relsda(access)(r13)    ; <== Volatile access
    or     r0,r0,r3
    cmpi   cr0,0,r0,3
    bne    cr0,.L2
    mr     r0,r7
.L2:
    bseti  r0,16
    li     r7,1
    stw    r7,@relsda(access)(r13)    ; <== Volatile access
    stw    r0,@relsda(variable)(r13)  ; <== Moved across volatile access
    blr
```

Result with **--language=+volatile**:

```
TestFunc:  .type func
    li     r0,0
    stw    r0,@relsda(access)(r13)    ; <== Volatile access
    lwz    r7,@relsda(variable)(r13)
    or     r7,r7,r3
    stw    r7,@relsda(variable)(r13)
    cmpi   cr0,0,r7,3
    bne    cr0,.L2
    stw    r0,@relsda(variable)(r13)
.L2:
    lwz    r0,@relsda(variable)(r13)
    bseti  r0,16
    stw    r0,@relsda(variable)(r13)
    li     r0,1
    stw    r0,@relsda(access)(r13)    ; <== Volatile access
    blr
```

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)

C compiler option: --make-target

Command line syntax

`--make-target=`*`name`*

Description

With this option you can overrule the default target name in the make dependencies generated by the options **--preprocess=+make** (**-Em**) and **--dep-file**. The default target name is the basename of the input file, with extension `.o`.

Example

```
cppc --preprocess=+make --make-target=mytarget.o test.c
```

The compiler generates dependency lines with the default target name `mytarget.o` instead of `test.o`.

Related information

C compiler option **--preprocess=+make** (Generate dependencies for make)

C compiler option **--dep-file** (Generate dependencies in a file)

C compiler option: --max-call-depth

Command line syntax

`--max-call-depth=`*`value`*

Default: -1

Description

This option is related to the compiler optimization **--optimize=+compact** (Code compaction or reverse inlining). Code compaction is the opposite of inlining functions: large sequences of code that occur more than once, are transformed into a function. This reduces code size (possibly at the cost of execution speed).

During code compaction it is possible that the compiler generates nested calls. This may cause the program to run out of its stack. To prevent stack overflow caused by too deeply nested function calls, you can use this option to limit the call depth. This option can have the following values:

-1	Poses no limit to the call depth (default)
0	The compiler will not generate any function calls. (Effectively the same as if you turned 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.)

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

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.

Related information

Section 3.1, *Compilation Process*

Control program option **--mil-link** / **--mil-split**

C compiler option: --misrac

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

```
cppc --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 3.8.2, *C Code Checking: MISRA-C*

C compiler option **--misrac-advisory-warnings**

C compiler option **--misrac-required-warnings**

Linker option **--misrac-report**

C compiler option: --misrac-advisory-warnings / --misrac-required-warnings

Command line syntax

`--misrac-advisory-warnings`

`--misrac-required-warnings`

Description

Normally, if an advisory rule or required rule is violated, the compiler generates an error. As a consequence, no output file is generated. With this option, the compiler generates a warning instead of an error.

Related information

Section 3.8.2, *C Code Checking: MISRA-C*

C compiler option **--misrac**

Linker option **--misrac-report**

C compiler option: --misrac-version

Command line syntax

`--misrac-version=``{`**`1998`**`|`**`2004`**`}`

Default: 2004

Description

MISRA-C rules exist in two versions: MISRA-C:1998 and MISRA-C:2004. By default, the C source is checked against the MISRA-C:2004 rules. With this option you can specify to check against the MISRA-C:1998 rules.

Related information

Section 3.8.2, *C Code Checking: MISRA-C*

C compiler option **--misrac**

C compiler option: --no-clear

Command line syntax

`--no-clear`

Description

Normally non-initialized global/static variables are cleared at program startup. With option **--no-clear** you tell the compiler to generate code to prevent non-initialized global/static variables from being cleared at program startup.

This option applies to constant as well as non-constant variables.

Related information

Pragmas `clear`/`noclear`

C compiler option: --no-double (-F)

Command line syntax

`--no-double`

`-F`

Description

With this option you tell the compiler to treat variables and constants of type `double` as `float`. Because the float type takes less space, execution speed increases and code size decreases, both at the cost of less precision.

Example

```
cppc --no-double test.c
```

The file `test.c` is compiled where variables and constants of type double are treated as float.

Related information

-

C compiler option: --no-fpu

Command line syntax

`--no-fpu`

Description

By default, the floating-point unit (FPU) is used if the selected core supports one. If an FPU is present, the macro `__FPU__` is defined in the C source file. Use this option to disable the use of the FPU.

Functions that have the `__fpu` function qualifier are not affected by this option. You can also disable the FPU for specific functions by using the `__nofpu` function qualifier.

Example

To disable the use of floating-point unit (FPU) instructions in the assembly code, enter:

```
cppc --no-fpu test.c
```

Related information

Section 1.8.5, *Floating-Point Unit Support: __fpu, __nofpu*

C compiler option **--fpu-version** (Select FPU version)

C compiler option: --no-macs

Command line syntax

```
--no-macs
```

Description

By default the compiler uses floating-point multiply-accumulate instructions. Use this option to disable the generation of these instructions.

Related information

-

C compiler option: --no-stdinc

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 3.4, *How the Compiler Searches Include Files*

C compiler option: --no-vle

Command line syntax

```
--no-vle
```

Description

By default VLE instructions are used if the selected core supports them. If VLE instructions are allowed, the macro `__VLE__` is defined in the C source file. Use this option to disable the generation of VLE instructions.

Functions that have the `__vle` function qualifier are not affected by this option. You can also disable VLE instructions for specific functions by using the `__novle` function qualifier.

Related information

Section 1.8.4, *VLE Instruction Support: __vle, __novle*

C compiler option: --no-warnings (-w)

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:

```
cppc test.c --no-warnings=537,538
```

Related information

C compiler option **--warnings-as-errors** (Treat warnings as errors)

Pragma `warning`

C compiler option: --optimize (-O)

Command line syntax

```
--optimize[=flags]
```

```
-Oflags
```

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
+/-wrap	**h/H**	Shrink wrapping
+/-inline	**i/I**	Automatic function inlining
+/-schedule	**k/K**	Instruction scheduler
+/-loop	**l/L**	Loop transformations
+/-align-loop	**n/N**	Align loops
+/-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
+/-peephole	**y/Y**	Peephole optimizations

Use the following options for predefined sets of flags:

--optimize=0	**-O0**	No optimization Alias for **-OaBCEFGHIKLNOPRSUY**

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 **-OabcefgHIKLNOPRSUy**

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 **-OabcefghIklNoprsUy**

Enables more optimizations to reduce code size and/or execution time. This is the default optimization level.

--optimize=3	**-O3**	Optimize most Alias for **-OabcefghiklNoprsuy**

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:

```
cppc test.c

cppc --optimize=2 test.c
cppc -O2 test.c

cppc --optimize test.c
cppc -O test.c

cppc -OabcefghIlNoprsy test.c
cppc --optimize=+coalesce,+ipro,+cse,+expression,+flow,+glo,
      +wrap,-inline,+schedule,+loop,-align-loop,+forward,
      +compact,+propagate,+subscript,-unroll,+peephole test.c
```

Related information

C compiler option **--tradeoff** (Trade off between speed and size)

Pragma `optimize/endoptimize`

Section 3.6, *Compiler Optimizations*

C compiler 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 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:

```
cppc --option-file=myoptions
```

This is equivalent to the following command line:

```
cppc --debug-info --define=DEMO=1 test.c
```

Related information

-

C compiler option: --output (-o)

Command line syntax

`--output=`*`file`*

`-o` *`file`*

Description

With this option you can specify another filename for the output file of the compiler. Without this option the basename of the C source file is used with extension `.src`.

Example

To create the file `output.src` instead of `test.src`, enter:

```
cppc --output=output.src test.c
```

Related information

-

C compiler option: --preprocess (-E)

Command line syntax

--preprocess[=*flags*]

-E[*flags*]

You can set the following flags:

+/-comments	**c/C**	keep comments
+/-includes	**i/I**	generate a list of included source files
+/-list	**l/L**	generate a list of macro definitions
+/-make	**m/M**	generate dependencies for make
+/-noline	**p/P**	strip #line source position information

Default: **-ECILMP**

Description

With this option you tell the compiler to preprocess the C source. On the command line, the compiler sends the preprocessed file to stdout. To capture the information in a file, specify an output file with the option **--output**.

With **--preprocess=+comments** you tell the preprocessor to keep the comments from the C source file in the preprocessed output.

With **--preprocess=+includes** the compiler will generate a list of all included source files. The preprocessor output is discarded.

With **--preprocess=+list** the compiler will generate a list of all macro definitions. The preprocessor output is discarded.

With **--preprocess=+make** the compiler will generate dependency lines that can be used in a Makefile. The preprocessor output is discarded. The default target name is the basename of the input file, with the extension `.o`. With the option **--make-target** you can specify a target name which overrules the default target name.

With **--preprocess=+noline** you tell the preprocessor to strip the #line source position information (lines starting with `#line`). These lines are normally processed by the assembler and not needed in the preprocessed output. When you leave these lines out, the output is easier to read.

Example

```
cppc --preprocess=+comments,-make,-noline test.c --output=test.pre
```

The compiler preprocesses the file `test.c` and sends the output to the file `test.pre`. Comments are included but no dependencies are generated and the line source position information is not stripped from the output file.

Related information

C compiler option **--dep-file** (Generate dependencies in a file)

C compiler option **--make-target** (Specify target name for **-Em** output)

C compiler option: --rename-sections (-R)

Command line syntax

--rename-sections[**=**[*type*[**.***attribute*]**=**][*format_string*]],...

-R[*type*[**.***attribute*]**=**]*format_string*,...

Default section name: .*type*.{module}.{name}

Description

By default the compiler extends the standard ELF section names with the module name and the name of the symbol that is allocated in the section. You can use this option to create your own unique section names to ease selection in linker script files for locating.

With the *type* and *attribute* you can select which sections will be renamed. When the type and attributes of a section match, the section name will get the specified *format string* as suffix. The following section types are allowed: "rodata", "data", "bss", "sdata", "sbss", "sdata0", "sbss0", "sdata2", "sbss2", "text", "text_vle" and "all". You cannot use the "all" section type in combination with an attribute.

The following attributes are allowed: init, noclear, romdata.

When you specify an optional attribute, only sections that have the attribute will be renamed. When you do not specify an attribute, only sections that do not have any of the listed attributes will be renamed. Note that the listed attributes are mutually exclusive; if a section uses one of the attributes, the other attributes will not be used.

When the type and attribute are omitted or type "all" is used, all sections will be renamed.

With the *format_string* you specify the string that extends the ELF section name. The format string can contain characters and may contain the following format specifiers:

`{attrib}`	section attributes, separated by underscores
`{module}`	module name
`{name}`	object name, name of variable or function

In format specifier expansions, dots are replaced with dollars ($).

When the *format_string* is omitted, only the section type will be used as the section name.

Example

To rename sections of memory type `data` to `.data.cppc.`*variable_name*:

```
cppc --rename-sections=data=cppc.{name} test.c
```

To generate the section name .*type*`.NEW` instead of the default section name .*type.module_name.symbol_name*, enter:

```
cppc -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:

```
cppc -R test.c
```

Related information

Section 1.9, *Section Naming*

C compiler option: --runtime (-r)

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: --schar

Command line syntax

`--schar`

Description

By default `char` is the same as specifying `unsigned char`. With this option `char` is the same as `signed char`.

Related information

Section 1.1, *Data Types*

C compiler option: --signed-bitfields

Command line syntax

```
--signed-bitfields
```

Description

For bit-fields it depends on the implementation whether a plain `int` is treated as `signed int` or `unsigned int`. By default an `int` bit-field is treated as `unsigned int`. This offers the best performance. With this option you tell the compiler to treat `int` bit-fields as `signed int`. In this case, you can still add the keyword `unsigned` to treat a particular `int` bit-field as `unsigned`.

Related information

Section 1.1, *Data Types*

C compiler option: --slow-unaligned-access

Command line syntax

```
--slow-unaligned-access
```

Description

On some processors unaligned data accesses can result in alignment exceptions. If that is the case, you can turn of unaligned data accesses with this option.

Related information

-

C compiler option: --small-enumeration

Command line syntax

```
--small-enumeration
```

Description

Normally the compiler treats enumerated types as `int`. With this option the compiler treats enum-types as `char` or `short` instead of `int` if the range of values for the enumeration permits this. This reduces memory footprint.

Related information

Section 1.1, *Data Types*

C compiler option: --source (-s)

Command line syntax

`--source`

`-s`

Description

With this option you tell the compiler to merge C source code with generated assembly code in the output file. The C source lines are included as comments.

Related information

Pragmas `source/nosource`

C compiler option: --stdout (-n)

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: --string-literal-memory

Command line syntax

`--string-literal-memory=`*`space`*

You can specify the following *space* arguments:

- **__data**
- **__sdata**
- **__sdata2**
- **__sdata0**

Default: **__data**

Description

With this option you can control the allocation of string literals. String literals are allocated in `__data` by default.

In the context of this option, a string literal used to initialize an array, as in:

```
char array[] = "string";
```

is not considered a string literal; i.e. this is an array initializer written as a string, equivalent to:

```
char array[] = { 's', 't', 'r', 'i', 'n', 'g', '\0' };
```

Strings literals as used in:

```
char * s = "string";
```

or:

```
printf( "formatter %s\n", "string" );
```

are affected by this option.

Example

To allocate string literals in `__sdata2` memory:

```
cppc --string-literal-memory=__sdata2 test.c
```

Related information

Pragma `string_literal_memory`

C compiler option **--constant-data-memory** (Assign memory to constant data objects)

C compiler option **--data-memory** (Assign memory to non-constant data objects)

C compiler option: --switch

Command line syntax

`--switch==`*`arg`*

You can give one of the following arguments:

auto	Choose most optimal code
jumptab	Generate jump tables
linear	Use linear jump chain code
lookup	Generate lookup tables

Default: **`--switch=auto`**

Description

With this option you tell the compiler which code must be generated for a switch statement: a jump chain (linear switch), a jump table or a lookup table. By default, the compiler will automatically choose the most efficient switch implementation based on code and data size and execution speed. This depends on the option **--tradeoff**.

Instead of this option you can use the following pragma:

```
#pragma switch arg
```

Example

To use a table filled with target addresses for each possible switch value, enter:

```
cppc --switch=jumptab test.c
```

Related information

Section 1.7, *Switch Statement*

C compiler option **--tradeoff** (Trade off between speed and size)

C compiler option: --tradeoff (-t)

Command line syntax

`--tradeoff=``{`**`0`**`|`**`1`**`|`**`2`**`|`**`3`**`|`**`4`**`}`

`-t``{`**`0`**`|`**`1`**`|`**`2`**`|`**`3`**`|`**`4`**`}`

Default: **`--tradeoff=4`**

Description

If the compiler uses certain optimizations (option **--optimize**), you can use this option to specify whether the used optimizations should optimize for more speed (regardless of code size) or for smaller code size (regardless of speed).

By default the compiler optimizes for code size (**--tradeoff=4**).

If you have not specified the option **--optimize**, the compiler uses the default *Optimize more* optimization. In this case it is still useful to specify a trade-off level.

Example

To set the trade-off level for the used optimizations:

```
cppc --tradeoff=2 test.c
```

The compiler uses the default *Optimize more* optimization level and balances speed and size while optimizing.

Related information

C compiler option **--optimize** (Specify optimization level)

Section 3.6.3, *Optimize for Size or Speed*

C compiler option: --undefine (-U)

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__`:

```
cppc --undefine=__TASKING__ test.c
```

Related information

C compiler option **--define** (Define preprocessor macro)

Section 1.6, *Predefined Preprocessor Macros*

C compiler option: --unroll-factor

Command line syntax

`--unroll-factor=`*`value`*

Default: **`--unroll-factor=-1`**

Description

With the loop unrolling optimization, short loops are eliminated by replacing them with a number of copies to reduce the number of branches. With this option you specify how many times eligible loops should be unrolled. When the unroll factor is -1 (default), small loops are unrolled automatically if the loop unrolling optimization (**--optimize=+unroll** / **-Ou**) is enabled and the optimization trade-off is set for speed (**--tradeoff=0** / **-t0**)).

Instead of this option you can use the following pragmas:

```
#pragma unroll_factor value
  ...
#pragma endunroll_factor
```

Example

To allow an unroll factor of four, enter:

```
cppc --optimize=+unroll --unroll-factor=4 --tradeoff=0 test.c
```

Related information

Pragma `unroll_factor`

C compiler option **--optimize** (Specify optimization level)

C compiler option **--tradeoff** (Trade off between speed and size)

Section 3.6, *Compiler Optimizations*

C compiler option: --version (-V)

Command line syntax

`--version`

`-V`

Description

Display version information. The compiler ignores all other options or input files.

Example

```
cppc --version
```

The compiler does not compile any files but displays the following version information:

```
TASKING VX-toolset for Power Architecture: C compiler   vx.yrz Build nnn
Copyright 2010-year Altium BV                    Serial# 00000000
```

Related information

-

C compiler option: --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 also 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`

7.2. Assembler Options

This section lists all assembler options.

Assembler option: --case-insensitive (-c)

Command line syntax

`--case-insensitive`

`-c`

Default: case sensitive

Description

With this option you tell the assembler not to distinguish between upper and lower case characters. By default the assembler considers upper and lower case characters as different characters.

Assembly source files that are generated by the compiler must always be assembled case sensitive. When you are writing your own assembly code, you may want to specify the case insensitive mode.

Example

When assembling case insensitive, the label `LabelName` is the same label as `labelname`.

```
asppc --case-insensitive test.src
```

Related information

-

Assembler option: --check

Command line syntax

```
--check
```

Description

With this option you can check the source code for syntax errors, without generating code. This saves time in developing your application.

The 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

Command line syntax

`--core=`*`core`*

You can specify the following *core* arguments:

e200z0	e200z0 core
e200z3	e200z3 core
e200z4	e200z4 core
e200z6	e200z6 core
e200z7	e200z7 core

Default: **e200z0**

Description

With this option you specify the core architecture for a target processor for which you create your application.

The macro `__CORE_E200Z0__` is set to 1.

Example

To allow the use of e200z6 instructions in the assembly code, enter:

```
asppc --core=e200z6 test.src
```

Related information

-

Assembler option: --debug-info (-g)

Command line syntax

`--debug-info``[=`*`flags`*`]`

`-g``[`*`flags`*`]`

You can set the following flags:

+/-asm	**a/A**	Assembly source line information
+/-hll	**h/H**	Pass high level language debug information (HLL)
+/-local	**l/L**	Assembler local symbols debug information
+/-smart	**s/S**	Smart debug information

Default: **`--debug-info=+hll`**

Default (without flags): **`--debug-info=+smart`**

Description

With this option you tell the assembler which kind of debug information to emit in the object file.

You cannot specify **--debug-info=+asm,+hll**. Either the assembler generates assembly source line information, or it passes HLL debug information.

When you specify **--debug-info=+smart**, the assembler selects which flags to use. If high level language information is available in the source file, the assembler passes this information (same as **--debug-info=-asm,+hll,-local**). If not, the assembler generates assembly source line information (same as **--debug-info=+asm,-hll,+local**).

With **--debug-info=AHLS** the assembler does not generate any debug information.

Related information

-

Assembler option: --define (-D)

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, 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 assembler with the option **--option-file** (**-f**) *file*.

Defining macros with this option (instead of in the assembly source) is, for example, useful in combination with conditional assembly as shown in the example below.

This option has the same effect as defining symbols via the `.DEFINE`, `.SET`, and `.EQU` directives. (similar to `#define` in the C language). With the `.MACRO` directive you can define more complex macros.

Example

Consider the following assembly program with conditional code to assemble a demo program and a real program:

```
.IF DEMO == 1
...          ; instructions for demo application
.ELSE
...          ; instructions for the real application
.ENDIF
```

You can now use a macro definition to set the DEMO flag:

```
asppc --define=DEMO test.src
asppc --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

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

```
asppc --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 **--make-target** (Specify target name for **--dep-file** output)

Assembler option: --diag

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:

```
asppc --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:

```
asppc --diag=html:all > aserrors.html
```

Related information

Section 4.6, *Assembler Error Messages*

Assembler option: --dwarf-version

Command line syntax

`--dwarf-version=``{`**`2`**`|`**`3`**`}`

Default: **`3`**

Description

With this option you tell the assembler which DWARF debug version to generate, DWARF2 or DWARF3 (default).

Related information

Section 10.1, *ELF/DWARF Object Format*

Assembler option: --emit-locals

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=+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: --endianness

Command line syntax

`--endianness=`*`endianness`*

`--little-endian`

You can specify the following *endianness*:

big	**b**	Big endian (default)
little	**l**	Little endian

Description

By default, the assembler generates object files with instructions and data in big-endian format (most significant byte of a word at lowest byte address). With **--endianness=little** the assembler generates object files in little-endian format (least significant byte of a word at lowest byte address). **--little-endian** is an alias for option **--endianness=little**.

The endianness is reflected in the list file.

Assembly code can check the setting of this option by means of the built-in assembly function `@BIGENDIAN()`.

Related information

Assembly function `@BIGENDIAN()`

Assembler option: --error-file

Command line syntax

`--error-file``[=`*`file`*`]`

Description

With this option the assembler redirects error messages to a file. If you do not specify a filename, the error file will be named after the input file with extension `.ers`.

Example

To write errors to `errors.ers` instead of `stderr`, enter:

```
asppc --error-file=errors.ers test.src
```

Related information

Section 4.6, *Assembler Error Messages*

Assembler option: --error-limit

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 4.6, *Assembler Error Messages*

Assembler option: --fpu

Command line syntax

`--fpu`

Description

With this option you can use floating-point unit (FPU) instructions in the assembly code.

Example

To allow the use FPU instructions in the assembly code, enter:

```
asppc --fpu test.src
```

Related information

Assembler option **--core** (Select core)

Assembler option: --help (-?)

Command line syntax

--help[=*item*]

-?

You can specify the following arguments:

options	Show extended option descriptions

Description

Displays an overview of all command line options. When you specify the argument **options** you can list detailed option descriptions.

Example

The following invocations all display a list of the available command line options:

```
asppc -?
asppc --help
asppc
```

To see a detailed description of the available options, enter:

```
asppc --help=options
```

Related information

-

Assembler option: --include-directory (-I)

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 `ASPPCINC` 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:

```
asppc --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)

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

```
asppc --include-file=myinc.inc test.src
```

The file `myinc.inc` is included at the beginning of `test.src` before it is assembled.

Related information

Assembler option **--include-directory** (Add directory to include file search path)

Assembler option: --keep-output-files (-k)

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)

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 source file with the extension `.lst`.

Related information

Assembler option **--list-format** (Format list file)

Assembler option: --list-format (-L)

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 (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

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

```
asppc --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: --no-macs

Command line syntax

```
--no-macs
```

Description

By default floating-point multiply-accumulate instructions are supported. Use this option to make these instructions invalid.

Related information

-

Assembler 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 201 and 202, enter:

```
asppc test.src --no-warnings=201,202
```

Related information

Assembler option **--warnings-as-errors** (Treat warnings as errors)

Assembler option: --optimize (-O)

Command line syntax

`--optimize=``flag,...`

`-O``flags`

You can set the following flags:

+/-generics	**g/G**	Allow generic instructions
+/-jumpchains	**j/J**	Optimize jump chains
+/-instr-size	**s/S**	Optimize instruction size

Default: **`--optimize=gJs`**

Description

With this option you can control the level of optimization. For details about each optimization see Section 4.4, *Assembler Optimizations*.

Related information

Section 4.4, *Assembler Optimizations*

Assembler 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 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=+asm,-local
test.src
```

Specify the option file to the assembler:

```
asppc --option-file=myoptions
```

This is equivalent to the following command line:

```
asppc --debug=+asm,-local test.src
```

Related information

-

Assembler option: --output (-o)

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:

```
asppc --output=relobj.o asm.src
```

Related information

-

Assembler option: --page-length

Command line syntax

`--page-length=`*`number`*

Default: 72

Description

If you generate a list file with the assembler option **--list-file**, this option sets the number of lines in a page in the list file. The default is 72, the minimum is 10. As a special case, a page length of 0 turns off page breaks.

Related information

Assembler option **--list-file** (Generate list file)

Assembler directive `.PAGE`

Assembler option: --page-width

Command line syntax

`--page-width=``number`

Default: 132

Description

If you generate a list file with the assembler option **--list-file**, this option sets the number of columns per line on a page in the list file. The default is 132, the minimum is 40.

Related information

Assembler option **--list-file** (Generate list file)

Assembler directive `.PAGE`

Assembler option: --preprocess (-E)

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)

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)

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:

```
asppc --list-file --section-info asm.src
```

Related information

Assembler option **--list-file** (Generate list file)

Assembler option: --symbol-scope (-i)

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 option: --version (-V)

Command line syntax

`--version`

`-V`

Description

Display version information. The assembler ignores all other options or input files.

Example

```
asppc --version
```

The assembler does not assemble any files but displays the following version information:

```
TASKING VX-toolset for Power Architecture: assembler   vx.yrz Build nnn
Copyright 2010-year Altium BV                 Serial# 00000000
```

Related information

-

Assembler option: --warnings-as-errors

Command line syntax

```
--warnings-as-errors[=number,...]
```

Description

If the assembler encounters an error, it stops assembling. When you use this option without arguments, you tell the assembler to treat all warnings as errors. This means that the exit status of the assembler will be non-zero after one or more assembler warnings. As a consequence, the assembler now also stops after encountering a warning.

You can also limit this option to specific warnings by specifying a comma-separated list of warning numbers.

Related information

Assembler option **--no-warnings** (Suppress some or all warnings)

7.3. Linker Options

This section lists all linker options.

Linker option: --case-insensitive

Command line syntax

`--case-insensitive`

Default: case sensitive

Description

With this option you tell the linker not to distinguish between upper and lower case characters in symbols. By default the linker considers upper and lower case characters as different characters.

Assembly source files that are generated by the compiler must *always* be assembled and thus linked case sensitive. When you have written your own assembly code and specified to assemble it case insensitive, you must also link the `.o` file case insensitive.

Related information

Assembler option **--case-insensitive**

Linker option: --chip-output (-c)

Command line syntax

`--chip-output=``[`*`basename`*`]:`*`format`*`[:`*`addr_size`*`],...`

`-c``[`*`basename`*`]:`*`format`*`[:`*`addr_size`*`],...`

You can specify the following formats:

IHEX	Intel Hex
SREC	Motorola S-records

The *addr_size* specifies the size of the addresses in bytes (record length). For Intel Hex you can use the values **1**, **2** or **4** bytes (default). For Motorola-S you can specify: **2** (S1 records), **3** (S2 records) or **4** bytes (S3 records, default).

Description

With this option you specify the Intel Hex or Motorola S-record output format for loading into a PROM-programmer. The linker generates a file for each ROM memory defined in the LSL file, where sections are located:

```
memory memname
{  type=rom;  }
```

The name of the file is the name of the memory device that was emitted with extension `.hex` or `.sre`. Optionally, you can specify a *basename* which prepends the generated file name.

The linker always outputs a debugging file in ELF/DWARF format and optionally an absolute object file in Intel Hex-format and/or Motorola S-record format.

Example

To generate Intel Hex output files for each defined memory, enter the following on the command line:

```
lkppc --chip-output=myfile:IHEX test.o
```

In this case, this generates the file `myfile_`*memname*`.hex`.

Related information

Linker option **--output** (Output file)

Linker option: --core (-C)

Command line syntax

`--core=`*`core`*

`-C`*`core`*

You can specify the following *core* arguments:

e200z0	PowerPC eSys e200z0 core
e200z4	PowerPC eSys e200z4 core
e200z6	PowerPC eSys e200z6 core
e200z7	PowerPC eSys e200z7 core

Default: **e200z0**

Description

With this option you specify the core architecture for the target processor for which you create your application.

In a multi-task setting, use this option to tell the linker to use a specific core for a specific task. Only one task can be assigned to a certain core. Assigning multiple tasks to a single core requires some form of kernel functionality.

Example

Specify a core:

```
lkppc --core=e200z6 test.o
```

Related information

-

Linker option: --define (-D)

Command line syntax

`--define=``*macro_name*[`**`=`**`*macro_definition*]`

`-D``*macro_name*[`**`=`**`*macro_definition*]`

Description

With this option you can define a macro and specify it to the linker LSL file preprocessor. If you only specify a macro name (no macro definition), the macro expands as '1'.

You can specify as many macros as you like; just use the option **--define** (**-D**) multiple times. If the command line exceeds the limit of the operating system, you can define the macros in an option file which you then must specify to the linker with the option **--option-file** (**-f**) *file*.

The definition can be tested by the preprocessor with `#if`, `#ifdef` and `#ifndef`, for conditional locating.

Example

To define the RESET vector, which is used in the linker script file `ppc_arch.lsl`, which is included in `default.lsl`, enter:

```
lkppc test.o -otest.elf --lsl-file=default.lsl --define=RESET=0x00000000
```

Related information

Linker option **--option-file** (Specify an option file)

Linker option: --diag

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:

```
lkppc --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:

```
lkppc --diag=html:all > lkerrors.html
```

Related information

Section 5.10, *Linker Error Messages*

Linker option: --endianness

Command line syntax

```
--endianness=endianness
```

```
--little-endian
```

You can specify the following *endianness*:

big	**b**	Big endian (default)
little	**l**	Little endian

Description

By default, the linker links objects in big-endian mode (most significant byte of a word at lowest byte address). With **--endianness=little** you tell the linker to link the input files in mode (least significant byte of a word at lowest byte address). The endianness used must be valid for the architecture you are linking for. Depending on the endianness used, the linker links different libraries.

--little-endian is an alias for option **--endianness=little**.

Related information

-

Linker option: --error-file

Command line syntax

`--error-file``[=`*`file`*`]`

Description

With this option the linker redirects error messages to a file. If you do not specify a filename, the error file is `lkppc.elk`.

Example

To write errors to `errors.elk` instead of `stderr`, enter:

```
lkppc --error-file=errors.elk test.o
```

Related information

Section 5.10, *Linker Error Messages*

Linker option: --error-limit

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 5.10, *Linker Error Messages*

Linker option: --extern (-e)

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:

```
lkppc 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`.

```
lkppc --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 5.3, *Linking with Libraries*

Linker option: --first-library-first

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:

```
lkppc --first-library-first a.a test.o b.a
```

If the file `test.o` calls a function which is both present in `a.a` and `b.a`, normally the function in `b.a` would be extracted. With this option the linker first tries to extract the symbol from the first library `a.a`.

Note that routines in `b.a` that call other routines that are present in both `a.a` and `b.a` are now also resolved from `a.a`.

Related information

Linker option **--no-rescan** (Rescan libraries to solve unresolved externals)

Linker option: --help (-?)

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:

```
lkppc -?
lkppc --help
lkppc
```

To see a detailed description of the available options, enter:

```
lkppc --help=options
```

Related information

-

Linker option: --hex-format

Command line syntax

`--hex-format=``*flag*,...`

You can set the following flag:

+/-start-address	**s/S**	Emit start address record

Default: **`--hex-format=s`**

Description

With this option you can specify to emit or omit the start address record from the hex file.

Related information

Linker option **--output** (Output file)

Section 10.2, *Intel Hex Record Format*

Linker option: --hex-record-size

Command line syntax

`--hex-record-size=`*`size`*

Default: 32

Description

With this option you can set the size (width) of the Intel Hex data records.

Related information

Linker option **--output** (Output file)

Section 10.2, *Intel Hex Record Format*

Linker option: --import-object

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 5.5, *Importing Binary Files*

Linker option: --include-directory (-I)

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:

```
lkppc --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)

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. `lkppc --incremental test1.o test2.o --output=test.out`

   test1.o and test2.o are linked

2. `lkppc --incremental test3.o test.out`

   test3.o and test.out are linked, task1.out is created

3. `lkppc task1.out`

   task1.out is located

Related information

Section 5.4, *Incremental Linking*

Linker option: --keep-output-files (-k)

Command line syntax

```
--keep-output-files
```

```
-k
```

Description

If an error occurs during linking, the resulting output file may be incomplete or incorrect. With this option you keep the generated output files when an error occurs.

By default the linker removes the generated output file when an error occurs. This is useful when you use the make utility. If the erroneous files are not removed, the make utility may process corrupt files on a subsequent invocation.

Use this option when you still want to use the generated file. For example when you know that a particular error does not result in a corrupt object file, or when you want to inspect the output file, or send it to Altium support.

Related information

Linker option **--warnings-as-errors** (Treat warnings as errors)

Linker option: --library (-l)

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 variable `LIBPPC`, unless you used the option **--ignore-default-library-path**.

Example

To search in the system library `libc.a` (C library):

```
lkppc 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 5.3, *Linking with Libraries*

Linker option: --library-directory (-L) / --ignore-default-library-path

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\e200`.

If you specify only **-L** (without a pathname) or the long option **--ignore-default-library-path**, the linker will not search the default path and also not in the paths specified in the environment variable `LIBPPC`. 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 variable `LIBPPC`.
3. The default directory `$(PRODDIR)\lib\e200`.

Example

Suppose you call the linker as follows:

```
lkppc 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 variable `LIBPPC`. Then the linker looks in the default directory `$(PRODDIR)\lib\e200` for libraries.

Related information

Linker option **--library** (Link system library)

Section 5.3.1, *How the Linker Searches Libraries*

Linker option: --link-only

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

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 5.7, *Controlling the Linker with a Script*

Linker option: --lsl-dump

Command line syntax

`--lsl-dump``[=`*`file`*`]`

Description

With this option you tell the linker to dump the LSL part of the map file in a separate file, independent of the option **--map-file** (generate map file). If you do not specify a filename, the file `lkppc.ldf` is used.

Related information

Linker option **--map-file-format** (Map file formatting)

Linker option: --lsl-file (-d)

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 5.7, *Controlling the Linker with a Script*

Linker option: --map-file (-M)

Command line syntax

`--map-file``[=`*`file`*`]`

`-M``[`*`file`*`]`

Default (linker): no map file is generated

Default (control program): 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`.

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.

The control program by default invokes the linker with the option **--map-file**.

Related information

Linker option **--map-file-format** (Format map file)

Section 9.2, *Linker Map File Format*

Linker option: --map-file-format (-m)

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)

Section 9.2, *Linker Map File Format*

Linker option: --misra-c-report

Command line syntax

`--misra-c-report[=file]`

Description

With this option you tell the linker to create a MISRA-C Quality Assurance report. This report lists the various modules in the project with the respective MISRA-C settings at the time of compilation. If you do not specify a filename, the file *basename*`.mcr` is used.

Related information

C compiler option **--misrac** (MISRA-C checking)

Linker option: --non-romable

Command line syntax

```
--non-romable
```

Description

With this option you tell the linker that the application must not be located in ROM. The linker will locate all ROM sections, including a copy table if present, in RAM. When the application is started, the data sections are re-initialized and the BSS sections are cleared as usual.

This option is, for example, useful when you want to test the application in RAM before you put the final application in ROM. This saves you the time of flashing the application in ROM over and over again.

Related information

-

Linker option: --no-rescan

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)

Command line syntax

```
--no-rom-copy
```

```
-N
```

Description

With this option the linker will not generate a ROM copy for data sections. A copy table is generated and contains entries to clear BSS sections. However, no entries to copy data sections from ROM to RAM are placed in the copy table.

The data sections are initialized when the application is downloaded. The data sections are not re-initialized when the application is restarted.

Related information

-

Linker option: --no-warnings (-w)

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:

```
lkppc --no-warnings=135,136 test.o
```

Related information

Linker option **--warnings-as-errors** (Treat warnings as errors)

Linker option: --optimize (-O)

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
+/-linktime-global-optimizations	**g/G**	Optimize loads (for speed) while linking
+/-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

Use the following options for predefined sets of flags:

--optimize=0	**-O0**	No optimization Alias for **-OCGLTX**
--optimize=1	**-O1**	Default optimization Alias for **-OcgLtx**
--optimize=2	**-O2**	All optimizations Alias for **-Ocgltx**

Default: **`--optimize=1`**

Description

With this option you can control the level of optimization.

Related information

For details about each optimization see Section 5.6, *Linker Optimizations*.

Linker 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 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:

```
lkppc --option-file=myoptions
```

This is equivalent to the following command line:

```
lkppc --map-file=my.map test.o --library-directory=c:\mylibs
```

Related information

-

Linker option: --output (-o)

Command line syntax

`--output=``[`*`filename`*`][:`*`format`*`[:`*`addr_size`*`][,`*`space_name`*`]]...`

`-o``[`*`filename`*`][:`*`format`*`[:`*`addr_size`*`][,`*`space_name`*`]]...`

You can specify the following formats:

ELF	ELF/DWARF
IHEX	Intel Hex
SREC	Motorola S-records

Description

By default, the linker generates an output file in ELF/DWARF format, with the name `task1.elf`.

With this option you can specify an alternative *filename*, and an alternative output *format*. The default output format is the format of the first input file.

You can use the **--output** option multiple times. This is useful to generate multiple output formats. With the first occurrence of the **--output** option you specify the basename (the filename without extension), which is used for subsequent **--output** options with no filename specified. If you do not specify a filename, or you do not specify the **--output** option at all, the linker uses the default basename `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).

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 `myfile.hex` of the address space named `linear`, enter:

```
lkppc test.o --output=myfile.hex:IHEX:2,linear
```

If they exist, any other address spaces are emitted as well and are named `myfile_`*spacename*`.hex`.

Related information

Linker option **--chip-output** (Generate an output file for each chip)

Linker option **--hex-format** (Specify Hex file format settings)

Linker option: --strip-debug (-S)

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)

Command line syntax

```
--user-provided-initialization-code
```

```
-i
```

Description

It is possible to use your own initialization code, for example, to save ROM space. With this option you tell the linker *not* to generate a copy table for initialize/clear sections. Use linker labels in your source code to access the positions of the sections when located.

If the linker detects references to the TASKING initialization code, an error is emitted: it is either the TASKING initialization routine or your own, not both.

Note that the options **--no-rom-copy** and **--non-romable**, may vary independently. The 'copytable-compression' optimization (**--optimize=t**) is automatically disabled when you enable this option.

Related information

Linker option **--no-rom-copy** (Do not generate ROM copy)

Linker option **--non-romable** (Application is not romable)

Linker option **--optimize** (Specify optimization)

Linker option: --verbose (-v) / --extra-verbose (-vv)

Command line syntax

```
--verbose / --extra-verbose
```

```
-v / -vv
```

Description

With this option you put the linker in *verbose* mode. The linker prints the link phases while it processes the files. In the *extra verbose* mode, the linker also prints the filenames and it shows which objects are extracted from libraries. With this option you can monitor the current status of the linker.

Related information

-

Linker option: --version (-V)

Command line syntax

`--version`

`-V`

Description

Display version information. The linker ignores all other options or input files.

Example

```
lkppc --version
```

The linker does not link any files but displays the following version information:

```
TASKING VX-toolset for Power Architecture: object linker   vx.yrz Build nnn
Copyright 2010-year Altium BV                      Serial# 00000000
```

Related information

-

Linker option: --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)

7.4. Control Program Options

The control program **ccppc** facilitates the invocation of the various components of the Power Architecture toolset from a single command line.

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, assembler or linker, it is recommended to use the control program options **--pass-c**, **--pass-assembler**, **--pass-linker**.

See the previous sections for details on the options of the tools.

Control program option: --address-size

Command line syntax

`--address-size=`*`addr_size`*

Description

If you specify IHEX or SREC with the control option **--format**, you can additionally specify the record length to be emitted in the output files.

With this option you can specify the size of 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:

```
ccppc --format=SREC --address-size=2 test.c
```

Related information

Control program option **--format** (Set linker output format)

Control program option **--output** (Output file)

Control program option: --case-insensitive

Command line syntax

`--case-insensitive`

Default: case sensitive

Description

With this option you tell the assembler not to distinguish between upper and lower case characters. By default the assembler considers upper and lower case characters as different characters.

Assembly source files that are generated by the compiler must always be assembled case sensitive. When you are writing your own assembly code, you may want to specify the case insensitive mode.

Example

When assembling case insensitive, the label `LabelName` is the same label as `labelname`.

```
ccppc --case-insensitive test.src
```

Related information

Assembler option **--case-insensitive**

Control program option: --check

Command line syntax

```
--check
```

Description

With this option you can check the source code for syntax errors, without generating code. This saves time in developing your application because the code will not actually be compiled.

The compiler/assembler reports any warnings and/or errors.

This option is available on the command line only.

Related information

C compiler option **--check** (Check syntax)

Assembler option **--check** (Check syntax)

Control program option: --core

Command line syntax

`--core=`*`core`*

You can specify the following *core* arguments:

e200z0	e200z0 core
e200z3	e200z3 core
e200z4	e200z4 core
e200z6	e200z6 core
e200z7	e200z7 core

Default: **e200z0**

Description

With this option you specify the core architecture for a target processor for which you create your application.

The macro `__CORE_E200Z0__` is defined in the C source file.

Example

Specify a core:

```
ccppc --core=e200z6 test.c
```

Related information

Control program option **--cpu** (Select processor)

Control program option **--cpu-list** (Show list of processors)

Control program option **--no-fpu** (Do not use FPU)

Control program option: --cpu (-C)

Command line syntax

`--cpu=`*`cpu`*

`-C`*`cpu`*

Description

With this option you define the target processor for which you create your application.

The standard list of supported processors is defined in the file `processors.xml`. This file defines for each processor its full name (for example, MPC5644A), its ID (for example, mpc5644a), the base CPU name (for example, e200z4) and the core settings (for example, e200z4).

The control program reads the file `processors.xml`. The lookup sequence for names specified to this option is as follows:

1. match any of the base CPU names (as listed above, for example `e200z4`)
2. if none matched, match with the '`id`' attribute in `processors.xml` (case insensitive, for example `mpc5644a`)
3. if still none matched, match with the '`name`' attribute in `processors.xml` (case insensitive, for example `MPC5644A`)
4. if still none matched, the control program issues a fatal error.

The control program passes the options to the underlaying tools. For example, **--core=e200z4** to the C compiler, or **-de200z4 -D__CPU__=e200z4** to the linker.

Example

To generate the file `test.elf` for the MPC5644A processor, enter:

```
ccppc --cpu=MPC5644A test.c
```

Related information

Control program option **--cpu-list** (Show list of processors)

Control program option **--processors** (Read additional processor definitions)

Control program option: --cpu-list

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:

```
ccppc --cpu-list
```

To show all processors that have mpc5644 in their name, enter:

```
ccppc --cpu-list=mpc5644

--- ~/cppc/etc/processors.xml ---
    id          name        CPU       core
    mpc5644a    MPC5644A    e200z4    e200z4
    mpc5644b    MPC5644B    e200z4    e200z4
    mpc5644c    MPC5644C    e200z4    e200z4
```

Related information

Control program option **--cpu** (Select processor)

Control program option: --create (-c)

Command line syntax

`--create``[=`*`stage`*`]`

`-c``[`*`stage`*`]`

You can specify the following stages:

relocatable	**l**	Stop after the files are linked to a linker object file (`.out`)
mil	**m**	Stop after C files are compiled to MIL (`.mil`)
object	**o**	Stop after the files are assembled to objects (`.o`)
assembly	**s**	Stop after 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`:

```
ccppc --create test.c
```

The control program stops after the file is assembled. It does not link nor locate the generated output.

Related information

Control program option **--link-only** (Link only, no locating)

Control program option: --debug-info (-g)

Command line syntax

```
--debug-info
```

```
-g
```

Description

With this option you tell the control program to include debug information in the generated object file.

The control program passes the option **--debug-info** (**-g**) to the C compiler and calls the assembler with **--debug-info=+smart,+local** (**-gsl**).

Related information

C compiler option **--debug-info** (Generate symbolic debug information)

Assembler option **--debug-info** (Generate symbolic debug information)

Control program option: --define (-D)

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, 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 compiler with the option **--option-file** (**-f**) *file*.

Defining macros with this option (instead of in the C source) is, for example, useful to compile conditional C source as shown in the example below.

The control program passes the option **--define** (**-D**) to the compiler and the assembler.

Example

Consider the following C program with conditional code to compile a demo program and a real program:

```
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:

```
ccppc --define=DEMO test.c
ccppc --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.

```
ccppc --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

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

```
ccppc --dep-file=test.dep -t test.c
```

The compiler compiles the file `test.c`, which results in the output file `test.src`, and generates dependency lines in the file `test.dep`.

Related information

Control program option **--preprocess=+make** (Generate dependencies for make)

Control program option: --diag

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 103, enter:

```
ccppc --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:

```
ccppc --diag=html:all > ccerrors.html
```

Related information

Section 3.9, *C Compiler Error Messages*

Control program option: --dry-run (-n)

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: --dwarf-version

Command line syntax

`--dwarf-version=``{`**`2`**`|`**`3`**`}`

Default: **3** (or **2** when **--eabi-compliant** is selected)

Description

With this option you tell the compiler which DWARF debug version to generate, DWARF2 or DWARF3 (default).

Note that only DWARF version 2 is EABI compliant.

Related information

Section 10.1, *ELF/DWARF Object Format*

Control program option: --eabi-compliant

Command line syntax

```
--eabi-compliant
```

Description

Use this option when the generated code needs to be completely EABI compliant.

This option is an alias for C compiler option **--eabi=DeLNRV**.

Related information

C compiler option **--eabi** (control level of EABI compliancy)

Control program option: --endianness

Command line syntax

`--endianness=`*`endianness`*

`--little-endian`

You can specify the following *endianness*:

big	**b**	Big endian (default)
little	**l**	Little endian

Description

By default, the compiler generates code for a big-endian target (most significant byte of a word at lowest byte address). With **--endianness=little** the compiler generates code for a little-endian target (least significant byte of a word at lowest byte address). **--little-endian** is an alias for option **--endianness=little**.

The endianness used must be a valid one for the architecture you are compiling for.

The control program passes this option to the C compiler, assembler and linker.

Related information

-

Control program option: --error-file

Command line syntax

```
--error-file
```

Description

With this option the control program tells the compiler, assembler and linker to redirect error messages to a file.

Example

To write errors to error files instead of stderr, enter:

```
ccppc --error-file test.c
```

Related information

Control Program option **--warnings-as-errors** (Treat warnings as errors)

Control program option: --format

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:

```
ccppc --format=SREC test1.c test2.c --output=test.sre
```

Related information

Control program option **--address-size** (Set address size for linker IHEX/SREC files)

Control program option **--output** (Output file)

Linker option **--chip-output** (Generate an output file for each chip)

Control program option: --fp-trap

Command line syntax

`--fp-trap`

Description

By default the control program uses the non-trapping floating-point library (`libfpv.a`). With this option you tell the control program to use the trapping floating-point library (`libfpvt.a`).

If you use the trapping floating-point library, exceptional floating-point cases are intercepted and can be handled separately by an application defined exception handler. Using this library decreases the execution speed of your application.

An example of the exception handler can be found in: `spe.c`.

Related information

Section 5.3, *Linking with Libraries*

Control program 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:

```
ccppc -?
ccppc --help
ccppc
```

To see a detailed description of the available options, enter:

```
ccppc --help=options
```

Related information

-

Control program option: --include-directory (-I)

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:

```
ccppc --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: --iso

Command line syntax

`--iso=``{`**`90`**`|`**`99`**`}`

Default: **`--iso=99`**

Description

With this option you select the ISO C standard. C90 is also referred to as the "ANSI C standard". C99 refers to the newer ISO/IEC 9899:1999 (E) standard. C99 is the default.

Independent of the chosen ISO standard, the control program always links libraries with C99 support.

Example

To select the ISO C90 standard on the command line:

```
ccppc --iso=90 test.c
```

Related information

C compiler option **--iso** (ISO C standard)

Control program option: --keep-output-files (-k)

Command line syntax

`--keep-output-files`

`-k`

Description

If an error occurs during the compilation, assembling or linking process, the resulting output file may be incomplete or incorrect. With this option you keep the generated output files when an error occurs.

By default the control program removes generated output files when an error occurs. This is useful when you use the make utility. If the erroneous files are not removed, the make utility may process corrupt files on a subsequent invocation.

Use this option when you still want to use the generated files. For example when you know that a particular error does not result in a corrupt file, or when you want to inspect the output file, or send it to Altium support.

The control program passes this option to the compiler, assembler and linker.

Example

```
ccppc --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)

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

```
ccppc --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)

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 variable `LIBPPC`, unless you used the option **--ignore-default-library-path**.

Example

To search in the system library `libcv.a` (C library):

```
ccppc test.o mylib.a --library=cv
```

The linker links the file `test.o` and first looks in library `mylib.a` (in the current directory only), then in the system library `libcv.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 5.3, *Linking with Libraries*

Control program option: --library-directory (-L) / --ignore-default-library-path

Command line syntax

--library-directory=*path*,...
-L*path*,...

--ignore-default-library-path
-L

Description

With this option you can specify the path(s) where your system libraries, specified with the option **--library** (**-l**), are located. If you want to specify multiple paths, use the option **--library-directory** for each separate path.

The default path is `$(PRODDIR)\lib\e200`.

If you specify only **-L** (without a pathname) or the long option **--ignore-default-library-path**, the linker will not search the default path and also not in the paths specified in the environment variable `LIBPPC`. 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 variable `LIBPPC`.
3. The default directory `$(PRODDIR)\lib\e200`.

Example

Suppose you call the control program as follows:

```
ccppc test.c --library-directory=c:\mylibs --library=cv
```

First the linker looks in the directory `c:\mylibs` for library `libcv.a` (this option). If it does not find the requested libraries, it looks in the directory that is set with the environment variable `LIBPPC`. Then the linker looks in the default directory `$(PRODDIR)\lib\e200` for libraries.

Related information

Control program option **--library** (Link system library)

Section 5.3.1, *How the Linker Searches Libraries*

Control program option: --link-only

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

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)

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 5.7, *Controlling the Linker with a Script*

Control program option: --make-target

Command line syntax

`--make-target=`*`name`*

Description

With this option you can overrule the default target name in the make dependencies generated by the options **--preprocess=+make** (**-Em**) and **--dep-file**. The default target name is the basename of the input file, with extension `.o`.

Example

```
ccppc --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: --mil-link / --mil-split

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.

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 3.1, *Compilation Process*

C compiler option **--mil** / **--mil-split**

Control program option: --no-default-libraries

Command line syntax

```
--no-default-libraries
```

Description

By default the control program specifies the standard C libraries (C99) and run-time library to the linker. With this option you tell the control program not to specify the standard C libraries and run-time library to the linker.

In this case you must specify the libraries you want to link to the linker with the option **--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

```
ccppc --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:

```
ccppc --no-default-libraries --library=c test.c
```

Related information

Control program option **--library** (Link system library)

Section 5.3.1, *How the Linker Searches Libraries*

Control program option: --no-fpu

Command line syntax

`--no-fpu`

Description

By default, the floating-point unit (FPU) is used if the selected core supports one. If an FPU is present, the macro `__FPU__` is defined in the C source file. Use this option to disable the use of the FPU.

Functions that have the `__fpu` function qualifier are not affected by this option. You can also disable the FPU for specific functions by using the `__nofpu` function qualifier.

Example

To disable the use of floating-point unit (FPU) instructions in the assembly code, enter:

```
ccppc --no-fpu test.c
```

Related information

Section 1.8.5, *Floating-Point Unit Support: __fpu, __nofpu*

Control program option **--use-double-precision-fp** (Do not replace doubles with floats)

Control program option: --no-macs

Command line syntax

```
--no-macs
```

Description

By default floating-point multiply-accumulate instructions are used. Use this option to disable the generation of these instructions.

Related information

-

Control program option: --no-map-file

Command line syntax

```
--no-map-file
```

Description

By default the control program tells the linker to generate a linker map file.

A linker map file is a text file that shows how the linker has mapped the sections and symbols from the various object files (`.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-vle

Command line syntax

`--no-vle`

Description

By default VLE instructions are used if the selected core supports them. Use this option to disable the generation of VLE instructions.

Functions that have the `__vle` function qualifier are not affected by this option. You can also disable VLE instructions for specific functions by using the `__novle` function qualifier.

The control program also tells the linker to link the corresponding C library.

Related information

Section 1.8.4, *VLE Instruction Support: __vle, __novle*

Control program option: --no-warnings (-w)

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:

```
ccppc test.c --no-warnings
```

Related information

Control program option **--warnings-as-errors** (Treat warnings as errors)

Control program 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 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:

```
ccppc --option-file=myoptions
```

This is equivalent to the following command line:

```
ccppc —-debug-info --define=DEMO=1 test.c
```

Related information

-

Control program option: --output (-o)

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

```
ccppc 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`:

```
ccppc --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)

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-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:

```
ccppc --pass-linker=--verbose test.c
```

Related information

-

Control program option: --preprocess (-E) / --no-preprocessing-only

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

Default: **`-ECILMP`**

Description

With this option you tell the compiler to preprocess the C source. The C compiler sends the preprocessed output to the file *name*`.pre` (where *name* is the name of the C source file to compile).

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.

Example

```
ccppc --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 **--make-target** (Specify target name for **-Em** output)

Control program option: --processors

Command line syntax

`--processors=`*`file`*

Description

With this option you can specify an additional XML file with processor definitions.

The standard list of supported processors is defined in the file `processors.xml`. This file defines for each processor its full name (for example, MPC5644A), its ID (for example, mpc5644a), the base CPU name (for example, e200z4) and the core settings (for example, e200z4).

The control program reads the specified *file* after the file `processors.xml` in the product's `etc` directory. Additional XML files can override processor definitions made in XML files that are read before.

Multiple **--processors** options are allowed.

Eclipse generates a **--processors** option in the makefiles for each specified XML file.

Example

Specify an additional processor definition file (suppose `processors-new.xml` contains a new processor `PPCNEW`):

```
ccppc --processors=processors-new.xml --cpu=PPCNEW test.c
```

Related information

Control program option **--cpu** (Select architecture)

Control program option: --static

Command line syntax

`--static`

Description

This option is directly passed to the compiler.

With this option, the compiler treats external definitions at file scope (except for `main`) as if they were declared `static`. As a result, unused functions will be eliminated, and the alias checking algorithm assumes that objects with static storage cannot be referenced from functions outside the current module.

This option only makes sense when you specify all modules of an application on the command line.

Example

```
ccppc --static module1.c module2.c module3.c ...
```

Related information

-

Control program option: --undefine (-U)

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__`:

```
ccppc --undefine=__TASKING__ test.c
```

Related information

Control program option **--define** (Define preprocessor macro)

Section 1.6, *Predefined Preprocessor Macros*

Control program option: --use-double-precision-fp

Command line syntax

`--use-double-precision-fp`

Description

When an FPU is present the control program will by default compile all doubles as floats to make full use of the FPU. When you do not want this, use the option **--use-double-precision-fp**.

Example

To allow the use of floating-point unit (FPU) instructions in the assembly code and treat 'double' as 'double', enter:

```
ccppc --use-double-precision-fp test.c
```

Related information

Control program option **--no-fpu** (Do not use FPU)

Control program option: --verbose (-v)

Command line syntax

```
--verbose
```

```
-v
```

Description

With this option you put the control program in verbose mode. The control program performs it 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)

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

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)

7.5. Parallel Make Utility Options

You can use the make utility **amk** 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 6.2, *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** | *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 169, enter:

```
amk --diag=169
```

This results in the following message and explanation:

```
F169: target '%s' returned exit code %d

An error 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

-?

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:

```
amk -?
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)

Command line syntax

```
--jobs[=number]
-j[number]
```

```
--jobs-limit[=number]
-J[number]
```

Description

When these options you can limit the number of parallel jobs. The default is 1. Zero means no limit. When you omit the *number*, **amk** uses the number of cores detected.

Option **-J** is the same as **-j**, except that the number of parallel jobs is limited by the number of cores detected.

Example

```
amk -j3
```

Limit the number of parallel jobs to 3.

Related information

-

Parallel make utility option: --keep-going (-k)

Command line syntax

`--keep-going`

`-k`

Description

When during the make process the make utility encounters an error, it stops rebuilding your files.

With the option **-k**, the make utility only stops building the target that produced the error. All other targets defined in the makefile are built.

Example

```
amk -k
```

If the make utility encounters an error, it stops building the current target but proceeds with the other targets that are defined in the makefile.

Related information

-

Parallel make utility option: --list-targets (-l)

Command line syntax

`--list-targets`

`-l`

Description

With this option, the make utility lists all "primary" targets that are out of date.

Example

```
amk -l
list of targets
```

Related information

-

Parallel make utility option: --makefile (-f)

Command line syntax

`--makefile=`*`my_makefile`*

`-f` *`my_makefile`*

Description

By default the make utility uses the file `makefile` to build your files.

With this option you tell the make utility to use the specified file instead of the file `makefile`. Multiple **-f** options act as if all the makefiles were concatenated in a left-to-right order.

If you use '-' instead of a makefile name it means that the information is read from `stdin`.

Example

```
amk -f mymake
```

The make utility uses the file `mymake` to build your files.

Related information

-

Parallel make utility option: --no-warnings (-w)

Command line syntax

```
--no-warnings[=number,...]
```

```
-w[number,...]
```

Description

With this option you can suppresses all warning messages or specific warning messages.

On the command line this option works as follows:

- If you do not specify this option, all warnings are reported.
- If you specify this option but without numbers, all warnings are suppressed.
- If you specify this option with a number, only the specified warning is suppressed. You can specify the option **--no-warnings=***number* multiple times.

Example

To suppress warnings 751 and 756, enter:

```
amk --no-warnings=751,756
```

Related information

Parallel make utility option **--warnings-as-errors** (Treat warnings as errors)

Parallel make utility option: --silent (-s)

Command line syntax

`--silent`

`-s`

Description

With this option you tell the make utility to perform its tasks without printing the commands it executes. Error messages are normally printed.

Example

```
amk -s
```

The make utility rebuilds your files but does not print the commands it executes during the make process.

Related information

Parallel make utility option **-n** (Perform a dry run)

Parallel make utility option: --version (-V)

Command line syntax

`--version`

`-V`

Description

Display version information. The make utility ignores all other options or input files.

Related information

-

Parallel make utility option: --warnings-as-errors

Command line syntax

`--warnings-as-errors``[=`*`number`*`,...]`

Description

If the make utility encounters an error, it stops. When you use this option without arguments, you tell the make utility to treat all warnings as errors. This means that the exit status of the make utility will be non-zero after one or more warnings. As a consequence, the make utility now also stops after encountering a warning.

You can also limit this option to specific warnings by specifying a comma-separated list of warning numbers.

Related information

Parallel make utility option **--no-warnings** (Suppress some or all warnings)

7.6. Archiver Options

The archiver and library maintainer **arppc** 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:

```
arppc 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 6.3, *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 -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*	
Create library without notification if library does not exist	**-c**	
Preserve last-modified date from the library	**-o**	
Print symbols in library modules	**-s**{**0**|**1**}	
Replace only newer modules	**-u**	
Verbose	**-v**	
Miscellaneous		

Description	Option	Sub-option
Display options	**-?**	
Display version header	**-V**	
Read options from *file*	**-f** *file*	
Suppress warnings above level *n*	**-w***n*	

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

```
arppc --delete mylib.a obj1.o obj2.o
```

The archiver deletes `obj1.o` and `obj2.o` from the library `mylib.a`.

```
arppc -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: --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

```
arppc --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`:

```
arppc --extract mylib.a obj1.o
```

If you do not specify an object module, all object modules are extracted:

```
arppc -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:

```
arppc -?
arppc --help
arppc
```

To see a detailed description of the available options, enter:

```
arppc --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`:

```
arppc --move mylib.a obj1.o
```

To move `obj3.o` just before `obj2.o`:

```
arppc -m -b obj3.o mylib.a obj2.o
```

The library `mylib.a` after these two invocations now looks like:

```
obj3.o
obj2.o
obj1.o
```

Related information

Archiver option **--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
-w5
```

Specify the option file to the archiver:

```
arppc --option-file=myoptions
```

This is equivalent to the following command line:

```
arppc -x mylib.a obj1.o -w5
```

Related information

-

Archiver option: --print (-t)

Command line syntax

```
--print [--symbols=0|1]

-t [-s0|-s1]
```

Description

Print a table of contents of the library to standard output. With the suboption **-s0** the archiver displays all symbols per object file.

--symbols=0	**-s0**	Displays per object the name of the object itself and all symbols in the object.
--symbols=1	**-s1**	Displays the symbols of all object files in the library in the form *library_name*:*object_name*:*symbol_name*

Example

```
arppc --print mylib.a
```

The archiver prints a list of all object modules in the library `mylib.a`:

```
arppc -t -s0 mylib.a
```

The archiver prints per object all symbols in the library. For example:

```
cstart.o
   symbols:
        _START
        __init_sp
        _start
        _endinit_clear
        _endinit_set
cinit.o
   symbols:
        _c_init
```

Related information

-

Archiver option: --replace (-r)

Command line syntax

```
--replace [--after=posname] [--before=posname][--create] [--newer-only] [--verbose]

-r [-a posname] [-b posname][-c] [-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**	Create a new library without checking whether it already exists. 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`:

```
arppc --replace mylib.a obj2.o
```

To insert `obj3.o` just before `obj2.o`:

```
arppc -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:

```
arppc --replace obj1.o newlib.a
```

The archiver creates the library `newlib.a` and adds the object `obj1.o` to it.

To *create a new library file and overwrite an existing library*, add an object file and specify an existing library with the supoption **-c**:

```
arppc -r -c obj1.o mylib.a
```

The archiver overwrites the library `mylib.a` and adds the object `obj1.o` to it. The new library `mylib.a` only contains `obj1.o`.

Related information

Archiver option **--print** (**-t**) (Print library contents)

Archiver option: --version (-V)

Command line syntax

`--version`

`-V`

Description

Display version information. The archiver ignores all other options or input files.

Example

```
arppc -V
```

The archiver displays the version information but does not perform any tasks.

```
TASKING VX-toolset for Power Architecture: ELF archiver   vx.yrz Build nnn
Copyright 2010-year Altium BV                         Serial# 00000000
```

Related information

-

Archiver option: --warning (-w)

Command line syntax

`--warning=``*level*`

`-w``*level*`

Description

With this suboption you tell the archiver to suppress all warnings above the specified level. The level is a number between 0 - 9.

The level of a message is printed between parentheses after the warning number. If you do not use the **-w** option, the default warning level is 8.

Example

To suppress warnings above level 5:

```
arppc --extract --warning=5 mylib.a obj1.o
```

Related information

-

7.7. HLL Object Dumper Options

The high level language (HLL) dumper **hldumpppc** 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:

```
hldumpppc -FdhMsy test.elf
hldumpppc --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: --address-size (-A)

Command line syntax

`--address-size=`*`addr_size`*

`-A`*`addr_size`*

Default: 4

Description

With this option you can specify the size of the addresses in bytes.

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

By default all sections are included.

Related information

Section 6.4.2, *HLL Dump Output Format*

HLL object dumper option: --data-dump-format (-d)

Command line syntax

--data-dump-format[=*format*]

-d[*format*]

You can specify one of the following formats:

directives	**d**	Dump data as directives. A new directive will be generated for each symbol.
hex	**h**	Dump data as hexadecimal code with ASCII translation.

Default: **--data-dump-format=directives**

Description

With this option you can control the way data 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.

With option **--data-dump-format=hex**, no directives will be generated for data sections, but data sections are dumped as hexadecimal code with ASCII translation. This only applies to ROM sections. RAM sections will be represented with only a start address and a size indicator.

Example

```
hldumpppc -F2 --section=.rodata.hello.\$2\$str hello.elf

---------- Section dump ----------

    .section .rodata.hello.$2$str
    .org 000005c6
    .db 48,65,6c,6c,6f,2c,20,25,73,21,0a,00            ; Hello, %s!..
    .endsec

hldumpppc -F2 --section=.rodata.hello.\$2\$str --data-dump-format=hex hello.elf

---------- Section dump ----------

                          section 33 (.rodata.hello.$2$str):
000005c6 48 65 6c 6c 6f 20 25 73 21 0a 00                Hello %s!..
```

Related information

Section 6.4.2, *HLL Dump Output Format*

HLL object dumper option: --disassembly-intermix (-i)

Command line syntax

`--disassembly-intermix`

`-i`

Description

With this option the disassembly is intermixed with HLL source code. The source is searched for as described with option **--source-lookup-path**

Example

```
hldumpppc --disassembly-intermix --source-lookup-path=c:\mylib\src hello.elf
```

Related information

HLL object dumper option **--source-lookup-path**

HLL object dumper option: --dump-format (-F)

Command line syntax

--dump-format[=*flag*,...]

-F[*flag*],...

You can specify the following format flags:

+/-dump	**d/D**	Dump the contents of the sections in the object file. Code sections can be disassembled, data sections are dumped.
+/-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.
+/-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 **DHMSY** (nothing)
	1	Alias for **DhMSY** (only HLL symbols)
	2	Alias for **dHMSY** (only section contents)
	3	Alias for **dhmsy** (default, everything)

Default: **--dump-format=dhmsy**

Description

With this option you can control which parts of the dump output you want to see. By default, all parts are dumped.

1. Module list
2. Section list
3. Section dump (disassembly)
4. HLL symbol table
5. Assembly level symbol table

You can limit the number of sections that will be dumped with the options **--sections** and **--section-types**.

Related information

Section 6.4.2, *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:

+/-fullpath	**f/F**	Include the full path to the field level.

Default (no flags): **`--expand-symbols=F`**

Description

With this option you specify that all struct, union and array symbols are expanded with their fields in the HLL symbol dump.

Example

```
hldumpppc -F1 hello.elf

---------- HLL symbol table ----------

400000d0     20 struct               _dbg_request [dbg.c]

hldumpppc -e -F1 hello.elf

---------- HLL symbol table ----------

400000d0     20 struct               _dbg_request [dbg.c]
400000d0      4   int                  _errno
400000d4      1   unsigned char        nr
400000d8     12   union                u
400000d8      4     struct               exit
400000d8      4       int                  status
400000d8      8     struct               open
400000d8      4       const unsigned char * pathname
400000dc      2       unsigned short int   flags
   ...

hldumpppc -ef -F1 hello.elf

---------- HLL symbol table ----------

400000d0     20 struct               _dbg_request [dbg.c]
400000d0      4   int                  _dbg_request._errno
400000d4      1   unsigned char        _dbg_request.nr
400000d8     12   union                _dbg_request.u
400000d8      4     struct               _dbg_request.u.exit
```

```
400000d8      4       int                     _dbg_request.u.exit.status
400000d8      8     struct                  _dbg_request.u.open
400000d8      4       const unsigned char * _dbg_request.u.open.pathname
400000dc      2       unsigned short int    _dbg_request.u.open.flags
   ...
```

Related information

Section 6.4.2, *HLL Dump Output Format*

HLL object dumper option: --help (-?)

Command line syntax

`--help`

`-?`

Description

Displays an overview of all command line options.

Example

The following invocations all display a list of the available command line options:

```
hldumpppc -?
hldumpppc --help
hldumpppc
```

Related information

-

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: --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:

```
hldumpppc --option-file=myoptions
```

This is equivalent to the following command line:

```
hldumpppc --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

```
hldumpppc --output=dump.txt hello.elf
```

The HLL object dumper dumps the output in file `dump.txt`.

Related information

HLL object dumper option **--output-type**

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

By default all sections are included.

Related information

Section 6.4.2, *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:

```
hldumpppc --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

```
hldumpppc -F2 hello.elf

----------- Section dump ----------

                                   .section .text_vle.hello.main
0000050c 7060e000 main             e_lis          r3,0
00000510 1c6305c6                  e_add16i       r3,r3,0x5c6
00000514 508d80cc                  e_lwz          r4,-0x7f34(r13)
00000518 78000004                  e_b            0x51c
                                   .endsec

hldumpppc --symbols=none -F2 hello.elf

----------- Section dump ----------

                                   .section .text_vle.hello.main
0000050c 7060e000                  e_lis          r3,0
00000510 1c6305c6                  e_add16i       r3,r3,0x5c6
00000514 508d80cc                  e_lwz          r4,-0x7f34(r13)
00000518 78000004                  e_b            0x51c
                                   .endsec
```

Related information

Section 6.4.2, *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

```
hldumpppc --output-type=xml --output=hello.xml ../hello.elf
```

The field `<File name="../hello.elf">` is used in `hello.xml`.

```
hldumpppc --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 8. Libraries

This chapter contains an overview of all library functions that you can call in your C source. This includes all functions of the standard C library (ISO C99) and some functions of the floating-point library.

Section 8.1, *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 8.2, *C Library Reentrancy*, gives an overview of which functions are reentrant and which are not.

The following libraries are included in the Power Architecture toolset. The directory `lib\e200` contains all libraries and has sub-directories `z0`, `z3`, `z4`, `z6` and `z7` for core specific libraries. The control program **ccppc** automatically selects the appropriate libraries depending on the specified options.

C library

Libraries	Description

libc[v][l][s][f].a	C libraries
	Optional letter:
	v = VLE instructions (no compiler option **--no-vle**)
	l = little-endian (compiler option **--endianness=little**)
	s = single precision floating-point (compiler option **--no-double**)
	f = FPU support (no compiler option **--no-fpu**)

libfp[v][l][t].a	Floating-point libraries
	Optional letter:
	v = VLE instructions (no compiler option **--no-vle**)
	l = little-endian (compiler option **--endianness=little**)
	t = trapping (control program option **--fp-trap**)

librt[v][l].a	Run-time library
	Optional letter:
	v = VLE instructions (no compiler option **--no-vle**)
	l = little-endian (compiler option **--endianness=little**)

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.

8.1. 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.

8.1.1. assert.h

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

8.1.2. 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`)

8.1.3. cstart.h

The header file `cstart.h` controls the system startup code's general settings and register initializations. It contains defines and prototypes of functions that can be used for hardware configuration.

8.1.4. ctype.h and wctype.h

The header file `ctype.h` declares the following functions which take a character *c* as an integer type argument. The header file `wctype.h` declares parallel wide-character functions which take a character *c* of the `wchar_t` type as argument.

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 '.', ',', '!').

ctype.h	wctype.h	Description
`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.

8.1.5. dbg.h

The header file `dbg.h` contains the debugger call interface for file system simulation. It contains low level functions. This header file is not defined in ISO C99.

`_dbg_trap`	Low level function to trap debug events
`_argcv(const char *`*`buf`*`,size_t `*`size`*`)`	Low level function for command line argument passing

8.1.6. errno.h

`int errno`	External variable that holds implementation defined error codes.

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

`EPERM`	1	Operation not permitted
`ENOENT`	2	No such file or directory
`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

8.1.7. except.h

The header file `except.h` is for floating-point exception handling. See control program option **--fp-trap**.

8.1.8. 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)*

8.1.9. fenv.h

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

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

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

```
FE_DIVBYZERO      FE_INEXACT       FE_INVALID
FE_OVERFLOW       FE_UNDERFLOW     FE_ALL_EXCEPT
```

`fegetround`	Returns the current rounding direction, represented as one of the values of the rounding direction macros. (*Not implemented*)
`fesetround`	Sets the current rounding directions. (*Not implemented*)

Currently no rounding mode macros are implemented.

8.1.10. 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 8.1.17, *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 8.1.17, *math.h and tgmath.h*)

8.1.11. 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 * restrict nptr, wchar_t ** restrict endptr, int base)`	Convert wide string to maximum sized unsigned integer. (Compare `wcstoull`)

8.1.12. 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)*

8.1.13. 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  ^=
```

8.1.14. limits.h

Contains the sizes of integral types, defined as macros.

8.1.15. locale.h

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

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

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

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

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

```
struct lconv *localeconv( void )
```

Returns a pointer to type `struct lconv` with values appropriate for the formatting of numeric quantities according to the rules of the current locale. The `struct lconv` in this header file is conforming the ISO standard.

8.1.16. 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 8.1.25, *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.
`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.

8.1.17. 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`.

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

frexp, ldexp, modf, scalbn, scalbln

math.h			tgmath.h	Description
frexp	frexpf	frexpl	frexp	Splits a float x into fraction *f* and exponent *n*, so that: $f = 0.0$ or $0.5 \le \| f \| \le 1.0$ 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*.

math.h			tgmath.h	Description
`scalbn`	`scalbnf`	`scalbnl`	`scalbn`	Computes the result of `x*FLT_RADIX`^{n}. efficiently, not normally by computing `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`. (*Not implemented*)
`lrint`	`lrintf`	`lrintl`	`lrint`	Returns the rounded integer value as a `long int` according to the current rounding direction. See `fenv.h`. (*Not implemented*)
`llrint`	`lrintf`	`lrintl`	`llrint`	Returns the rounded integer value as a `long long int` according to the current rounding direction. See `fenv.h`. (*Not implemented*)
`nearbyint`	`nearbyintf`	`nearbyintl`	`nearbyint`	Returns the rounded integer value as a floating-point according to the current rounding direction. See `fenv.h`. (*Not implemented*)
`round`	`roundf`	`roundl`	`round`	Returns the nearest integer value of `x` as int. (*Not implemented*)
`lround`	`lroundf`	`lroundl`	`lround`	Returns the nearest integer value of `x` as long int. (*Not implemented*)
`llround`	`lroundf`	`llroundl`	`llround`	Returns the nearest integer value of `x` as long long int. (*Not implemented*)
`trunc`	`truncf`	`truncl`	`trunc`	Returns the truncated integer value `x`. (*Not implemented*)

Remainder after division

math.h			tgmath.h	Description
`fmod`	`fmodf`	`fmodl`	`fmod`	Returns the remainder `r` of `x-ny`. n is chosen as `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 `trunc(`*x/y*`)`. `r` may not have the same sign as `x`. (*Not implemented*)
`remquo`	`remquof`	`remquol`	`remquo`	Same as remainder. In addition, the argument `*quo` is given a specific value (see ISO). (*Not implemented*)

Power and absolute-value functions

math.h			tgmath.h	Description
`cbrt`	`cbrtf`	`cbrtl`	`cbrt`	Returns the real cube root of x (=$x^{1/3}$). (*Not implemented*)
`fabs`	`fabsf`	`fabsl`	`fabs`	Returns the absolute value of x (`|x|`). (`abs`, `labs`, `llabs`, `div`, `ldiv`, `lldiv` are defined in `stdlib.h`)
`fma`	`fmaf`	`fmal`	`fma`	Floating-point multiply add. Returns `x*y+z`. (*Not implemented*)
`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`	`copysignl`	`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`*. (*Not implemented*)
`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`. (*Not implemented*)
`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`. (*Not implemented*)

Positive difference, maximum, minimum

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

Error and gamma (Not implemented)

math.h			tgmath.h	Description
`erf`	`erff`	`erfl`	`erf`	Computes the error function of x. (*Not implemented*)

math.h			**tgmath.h**	**Description**
`erfc`	`erfcf`	`erfcl`	`erc`	Computes the complementary error function of x. (*Not implemented*)
`lgamma`	`lgammaf`	`lgammal`	`lgamma`	Computes the $*\log_e\lvert\Gamma(x)\rvert$ (*Not implemented*)
`tgamma`	`tgammaf`	`tgammal`	`tgamma`	Computes $\Gamma(x)$ (*Not implemented*)

Comparison macros

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

math.h	**tgmath.h**	**Description**
`isgreater`	-	Returns the value of `(x) > (y)`
`isgreaterequal`	-	Returns the value of `(x) >= (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 therefor do not have a parallel function in `tgmath.h`. All arguments must be expressions of real-floating type.

math.h	**tgmath.h**	**Description**
`fpclassify`	-	Returns the class of its argument: `FP_INFINITE`, `FP_NAN`, `FP_NORMAL`, `FP_SUBNORMAL` or `FP_ZERO`
`isfinite`	-	Returns a nonzero value if and only if its argument has a finite value
`isinf`	-	Returns a nonzero value if and only if its argument has an 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.

8.1.18. 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()`.

8.1.19. 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.

8.1.20. 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.

8.1.21. 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
```

8.1.22. 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 0 (zero).
`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`.

8.1.23. stdint.h

See Section 8.1.11, *inttypes.h and stdint.h*

8.1.24. 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 0 (zero).
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: `"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	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
r, lr	__fract, __lfract

Character	Printed as
R, lR	__accum, __laccum
%	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

Character	Scanned as
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.
r, lr	__fract, __lfract
R, lR	__accum, __laccum
[...]	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 8.1.20, *stdarg.h*)
`vscanf(`*`format`*`,` *`arg`*`)`	`vwscanf(`*`format`*`,` *`arg`*`)`	Same as `sscanf`/`swscanf`, but extra arguments are given as variable argument list *arg*. (See Section 8.1.20, *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 8.1.20, *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)*

stdio.h	wchar.h	Description
`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 8.1.20, *stdarg.h*) (*FSS implementation)*
`vprintf(format, arg)`	`vwprintf(format, arg)`	Same as `printf/wprintf`, but extra arguments are given as variable argument list *arg*. (See Section 8.1.20, *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 8.1.20, *stdarg.h*)

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.
`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, n, stdin)`	-	Reads at most the next *n*-1 characters from the `stdin` stream into array *s*. A newline is ignored. Returns *s* or NULL or EOF/WEOF on error. (*FSS implementation)*
`ungetc(c, stream)`	`ungetwc(c, stream)`	Pushes character *c* back onto the input *stream*. Returns EOF/WEOF on error.

stdio.h	wchar.h	Description
`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 8.1.6, *errno.h*)

8.1.25. 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 8.1.15, *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.
`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 8.1.19, *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.
`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 multi-byte character pointed to by *s*. At most *n* characters will be examined. (See also `mbrlen` in Section 8.1.29, *wchar.h*).
`mbtowc(*`*pwc*`,*`*s*`,`*n*`)`	Converts the multi-byte 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 multi-byte 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 multi-byte characters in the string pointed to by *s* into a sequence of wide characters and stores at most *n* wide characters into the array pointed to by *pwcs*. (See also `mbsrtowcs` in Section 8.1.29, *wchar.h*).
`wcstombs(*`*s*`,*`*pwcs*`,`*n*`)`	Converts a sequence of wide characters in the array pointed to by *pwcs* into multi-byte characters and stores at most *n* multi-byte characters into the string pointed to by *s*. (See also `wcsrtowmb` in Section 8.1.29, *wchar.h*).

8.1.26. 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.

string.h	wchar.h	Description
`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*.
`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 8.1.15, *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*.

string.h	wchar.h	Description
`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*.
`strerror(errno)`	-	Typically, the values for errno come from `int errno`. This function returns a pointer to the associated error message. (See also Section 8.1.6, *errno.h*)
`strlen(*s)`	`wcslen(*s)`	Returns the length of string **s*.

8.1.27. 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 is not implemented because it strongly depends on the hardware. To determine the time in seconds, the result of clock should be divided by the value defined by `CLOCKS_PER_SEC`.

`difftime(`*`t1`*`,`*`t0`*`)`	Returns the difference *t1-t0* in seconds.
`mktime(tm *`*`tp`*`)`	Converts the broken-down time in the structure pointed to by *tp*, to a value of type `time_t`. The return value has the same encoding as the return value of the `time` function.
`time(*`*`timer`*`)`	Returns the current calendar time. This value is also assigned to `*timer`.

Time conversion

`asctime(tm *`*`tp`*`)`	Converts the broken-down time in the structure pointed to by *tp* into a string in the form `Mon Jan 22 16:15:14 2007\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`*`)`	`wstrftime(*`*`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 8.1.15, *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
%%	%

8.1.28. 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)*

8.1.29. wchar.h

Many functions in `wchar.h` represent the wide-character variant of other functions so these are discussed together. (See Section 8.1.24, *stdio.h and wchar.h*, Section 8.1.25, *stdlib.h and wchar.h*, Section 8.1.26, *string.h and wchar.h* and Section 8.1.27, *time.h and wchar.h*).

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

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

```
                                    character <=> multibyte
                                    conversion */
} mbstate_t;
```

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

`mbsinit(*ps)`	Determines whether the object pointed to by *ps*, is an initial conversion state. Returns a non-zero value if so.
`mbsrtowcs(*pwcs,**src,n,*ps)`	Restartable version of `mbstowcs`. See Section 8.1.25, *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 8.1.25, *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 8.1.25, *stdlib.h and wchar.h*.
`wcrtomb(*s,wc,*ps)`	Converts a wide character wc to a multi-byte character according to conversion state *ps* and stores the multi-byte character in **s*.
`btowc(c)`	Returns the wide character corresponding to character *c*. Returns WEOF on error.
`wctob(c)`	Returns the multi-byte character corresponding to the wide character *c*. The returned multi-byte character is represented as one byte. Returns EOF on error.
`mbrlen(*s,n,*ps)`	Inspects up to *n* bytes from the string **s* to see if those characters represent valid multibyte characters, relative to the conversion state held in **ps*.

8.1.30. wctype.h

Most functions in `wctype.h` represent the wide-character variant of functions declared in `ctype.h` and are discussed in Section 8.1.4, *ctype.h and wctype.h*. In addition, this header file provides extensible, locale specific functions and wide character classification.

`wctype(*property)`	Constructs a value of type `wctype_t` that describes a class of wide characters identified by the string **property*. If property identifies a valid class of wide characters according to the LC_TYPE category (see Section 8.1.15, *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 8.1.15, *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")`
`towupper(`*`wc`*`)`	`towctrans(`*`wc`*`,wctrans("toupper")`

8.2. 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).

Function	Not reentrant because
`_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.
`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.
`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`	-
`cabs cabsf cabsl`	Sets errno via calls to other functions.
`cacos cacosf cacosl`	Sets errno via calls to other functions.
`cacosh cacosh cfacoshl`	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 casinh cfasinhl`	Sets errno via calls to other functions.

Function	Not reentrant because
`catan catanf catanl`	Sets errno via calls to other functions.
`catanh catanhf catanhl`	Sets errno via calls to other functions.
`cbrt cbrtf cbrtl`	(*Not implemented*)
`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`	Uses global File System Simulation buffer, _dbg_request
`clog clogf clogl`	Sets errno via calls to other functions.
`close`	Calls _close
`conj conjf conjl`	-
`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`	(*Not implemented*)
`erfc erfcf erfcl`	(*Not implemented*)
`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.

Function	Not reentrant because
`exp2 exp2f exp2l`	(*Not implemented*)
`expm1 expm1f expm1l`	(*Not implemented*)
`fabs fabsf fabsl`	-
`fclose`	Uses values in iob[ ]. See (1).
`fdim fdimf fdiml`	(*Not implemented*)
`feclearexcept`	(*Not implemented*)
`fegetenv`	(*Not implemented*)
`fegetexceptflag`	(*Not implemented*)
`fegetround`	(*Not implemented*)
`feholdexept`	(*Not implemented*)
`feof`	Uses values in iob[ ]. See (1).
`feraiseexcept`	(*Not implemented*)
`ferror`	Uses values in iob[ ]. See (1).
`fesetenv`	(*Not implemented*)
`fesetexceptflag`	(*Not implemented*)
`fesetround`	(*Not implemented*)
`fetestexcept`	(*Not implemented*)
`feupdateenv`	(*Not implemented*)
`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`	(*Not implemented*)
`fmax fmaxf fmaxl`	(*Not implemented*)
`fmin fminf fminl`	(*Not implemented*)
`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).

Function	Not reentrant because
`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`	(*Not implemented*)
`imaxabs`	-
`imaxdiv`	-
`isalnum iswalnum`	-
`isalpha iswalpha`	-
`isascii iswascii`	-
`iscntrl iswcntrl`	-
`isdigit iswdigit`	-
`isfinite`	-
`isgraph iswgraph`	-
`isgreater`	-
`isgreaterequal`	-
`isinf`	-
`isless`	-
`islessequal`	-
`islessgreater`	-
`islower iswlower`	-
`isnan`	-
`isnormal`	-
`isprint iswprint`	-
`ispunct iswpunct`	-

Function	Not reentrant because
`isspace iswspace`	-
`isunordered`	-
`isupper iswupper`	-
`iswalnum`	-
`iswalpha`	-
`iswcntrl`	-
`iswctype`	-
`iswdigit`	-
`iswgraph`	-
`iswlower`	-
`iswprint`	-
`iswpunct`	-
`iswspace`	-
`iswupper`	-
`iswxditig`	-
`isxdigit iswxdigit`	-
`ldexp ldexpf ldexpl`	Sets errno. See (2).
`lgamma lgammaf lgammal`	(*Not implemented*)
`llrint lrintf lrintl`	(*Not implemented*)
`llround llroundf llroundl`	(*Not implemented*)
`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`	(*Not implemented*)
`log2 log2f log2l`	(*Not implemented*)
`logb logbf logbl`	(*Not implemented*)
`longjmp`	-
`lrint lrintf lrintl`	(*Not implemented*)
`lround lroundf lroundl`	(*Not implemented*)
`lseek`	Calls _lseek
`lstat`	(*Not implemented*)
`malloc`	Needs kernel support. See (5).
`mblen`	N.A., skeleton function
`mbrlen`	Sets errno.

Function	Not reentrant because
`mbrtowc`	Sets errno.
`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`	-
`nan nanf nanl`	(*Not implemented*)
`nearbyint nearbyintf nearbyintl`	(*Not implemented*)
`nextafter nextafterf nextafterl`	(*Not implemented*)
`nexttoward nexttowardf nexttowardl`	(*Not implemented*)
`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`	-
`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`	(*Not implemented*)
`remove`	Uses global File System Simulation buffer, _dbg_request

Function	Not reentrant because
`remquo remquof remquol`	(*Not implemented*)
`rename`	Uses global File System Simulation buffer, _dbg_request
`rewind`	Eventually calls _lseek
`rint rintf rintl`	(*Not implemented*)
`round roundf roundl`	(*Not implemented*)
`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
`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 wstrftime`	-
`strlen wcslen`	-
`strncat wcsncat`	-
`strncmp wcsncmp`	-
`strncpy wcsncpy`	-
`strpbrk wcspbrk`	-

Function	Not reentrant because
`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).
`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`	(*Not implemented*)
`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`	(*Not implemented*)
`ungetc ungetwc`	Uses static buffer to hold unget characters for each file. Can be moved into iob structure. See (1).
`unlink`	Uses global File System Simulation buffer, _dbg_request
`vfprintf vfwprintf`	Uses iob[ ]. See (1).
`vfscanf vfwscanf`	Calls _doscan
`vprintf vwprintf`	Uses iob[ ]. See (1).

Function	Not reentrant because
`vscanf vwscanf`	Calls _doscan
`vsprintf vswprintf`	Sets errno.
`vsscanf vswscanf`	Sets errno.
`wcrtomb`	Sets errno.
`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`	-
`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 9. List File Formats

This chapter describes the format of the assembler list file and the linker map file.

9.1. 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 4.5, *Generating a List File*.

The list file consists of a page header and a source listing.

Page header

The page header is repeated on every page:

```
TASKING VX-toolset for PowerPC: assembler vx.yrz Build nnn SN 00000000
Title                                                            Page 1

ADDR CODE      CYCLES  LINE SOURCE LINE
```

The first line contains version information. The second line can contain a title which you can specify with the assembler directive `.TITLE` and always contains a page number. The third line is empty and the fourth line contains the headings of the columns for the source listing.

With the assembler directives `.LIST`/`.NOLIST`, `.PAGE`, and with the assembler option **--list-format** you can format the list file.

Source listing

The following is a sample part of a listing. An explanation of the different columns follows below.

```
ADDR CODE      CYCLES  LINE SOURCE LINE
                          1         ; Module start
                          .
                          .
0000                      8 main:   .type   func
0000 706rErrr  1    1     9         lis     r3,@ha(.2.str)
0004 1C63rrrr  1    2    10         la      r3,@lo(.2.str)(r3)
0008 508Drrrr  1    3    11         lwz     r4,@relsda(world)(r13)
000C 78rrrrrr  1    4    12         b       printf
                          .
                          .
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.

9.2. Linker Map File Format

The linker map file is an additional output file of the linker that shows how the linker has mapped the sections and symbols from the various object files (`.o`) to output sections. The locate part shows the absolute position of each section. External symbols are listed per space with their absolute address, both sorted on symbol and sorted on address. For details on how to generate a map file, see Section 5.9, *Generating a Map File*.

With the linker option **--map-file-format** you can specify which parts of the map file you want to see.

Tool and Invocation

This part of the map file contains information about the linker, its version header information, binary location and which options are used to call it.

Used Resources

This part of the map file shows the memory usage at memory level and space level. The largest free block of memory (`Largest gap`) is also shown. This part also contains an estimation of the stack usage.

Explanation of the columns:

Memory	The names of the memory as defined in the linker script file (`*.lsl`).
Code	The size of all executable sections.

Data	The size of all non-executable sections (not including stacks, heaps, debug sections in non-alloc space).
Reserved	The total size of reserved memories, reserved ranges, reserved special sections, stacks, heaps, alignment protections, sections located in non-alloc space (debug sections). In fact, this size is the same as the size in the Total column minus the size of all other columns.
Free	The free memory area addressable by this core. This area is accessible for unrestricted items.
Total	The total memory area addressable by this core.
Space	The names of the address spaces as defined in the linker script file (`*.lsl`). The names are constructed of the `derivative` name followed by a colon ':', the `core` name, another colon ':' and the `space` name. For example: `spe:ppc:linear`.
Native used ...	The size of sections located in this space.
Foreign used	The size of all sections destined for/located in other spaces, but because of overlap in spaces consume memory in this space.
Stack Name	The name(s) of the stack(s) as defined in the linker script file (`*.lsl`).
Used	An estimation of the stack usage. The linker calculates the required stack size by using information (`.CALLS` directives) generated by the compiler. If for example recursion is detected, the calculated stack size is inaccurate, therefore this is an estimation only. The calculated stack size is supposed to be smaller than the actual allocated stack size. If that is not the case, then a warning is given.

Processed Files

This part of the map file shows all processed files. This also includes object files that are extracted from a library, with the symbol that led to the extraction.

Link Result

This part of the map file shows per object file how the link phase has mapped the sections from the various object files (`.o`) to output sections.

[in] File	The name of an input object file.
[in] Section	A section name and id from the input object file. The number between '( )' uniquely identifies the section.
[in] Size	The size of the input section.
[out] Offset	The offset relative to the start of the output section.
[out] Section	The resulting output section name and id.
[out] Size	The size of the output section.

Module Local Symbols

This part of the map file shows a table for each local scope within an object file. Each table has three columns, 1 the symbol name, 2 the address of the symbol and 3 the space where the symbol resides in. The table is sorted on symbol name within each space.

By default this part is not shown in the map file. You have to turn this part on manually with linker option **--map-file-format=+statics** (module local symbols).

Cross References

This part of the map file lists all symbols defined in the object modules and for each symbol the object modules that contain a reference to the symbol are shown. Also, symbols that remain undefined are shown.

Call Graph

This part of the map file contains a schematic overview that shows how (library) functions call each other. To obtain call graph information, the assembly file must contain `.CALLS` directives.

Overlay

This part is empty for the PowerPC.

Locate Result: Sections

This part of the map file shows the absolute position of each section in the absolute object file. It is organized per address space, memory chip and group and sorted on space address.

#	The line number and default sort order.
Section **Section name** **Section number**	The name and id of the section. The number between '( )' uniquely identifies the section. Names within square brackets **[ ]** will be copied during initialization from ROM to the corresponding section name in RAM.
Sect. size (hex) **Sect. size (dec)**	The size of the section in minimum addressable units (hexadecimal or decimal).
Group	Sections can be ordered in groups. These are the names of the groups as defined in the linker script file (`*.lsl`) with the keyword `group` in the `section_layout` definition. The name that is displayed is the name of the deepest nested group.
Start address	The first address of the section in the address space.
End address	The last address of the section in the address space.
Symbols in sect.	The names of the external symbols that are referenced in the section. See **Locate Result: Symbols** below.
Defined in	The names of the input modules the section is defined in. See **Link Result: [in] File** above.
Referenced in	The names of the modules that contain a reference to the section. See **Cross References** above.

Chip name	The names of the memory chips as defined in the linker script file (`*.lsl`) in the `memory` definitions.
Chip addr	The absolute offset of the section from the start of a memory chip.
Locate type:properties	The locate rule type and properties. See **Locate Rules** below.

Locate Result: Symbols

This part of the map file lists all external symbols per address space name.

Address	The absolute address of the symbol in the address space.
Name	The name of the symbol.
Space	The names of the address spaces as defined in the linker script file (`*.lsl`). The names are constructed of the `derivative` name followed by a colon ':', the `core` name, another colon ':' and the `space` name. For example: `spe:ppc:linear`.

Processor and Memory

This part of the map file shows the processor and memory information of the linker script file.

By default this part is not shown in the map file. You have to turn this part on manually with linker option **--map-file-format=+lsl** (processor and memory info). You can print this information to a separate file with linker option **--lsl-dump**.

Locate Rules

This part of the map file shows the rules the linker uses to locate sections.

Address space The names of the address spaces as defined in the linker script file (`*.lsl`). The names are constructed of the `derivative` name followed by a colon ':', the `core` name, another colon ':' and the `space` name.

Type The rule type:

`ordered/contiguous/clustered/unrestricted`

Specifies how sections are grouped. By default, a group is 'unrestricted' which means that the linker has total freedom to place the sections of the group in the address space.

`absolute`

The section must be located at the address shown in the Properties column.

`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 10. Object File Formats

This chapter describes the format of several object files.

10.1. ELF/DWARF Object Format

The TASKING VX-toolset for Power Architecture by default produces objects in the ELF/DWARF 2 format.

The ELF/DWARF 2 Object Format for the Power Architecture toolset follows the convention as described in the *PowerPC Embedded Application Binary Interface* [Freescale Semiconductor, Inc.].

For a complete description of the ELF and DWARF formats, please refer to the *Tool Interface Standard (TIS)*.

10.2. 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 specify the Control program option **--format=IHEX**. Optionally specify the address size with Control program option **--address-size**.

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:

```
:0F00200000232222754E00754F04AF4FAE4E22C3
 | |   | |                             |_ 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:

```
:04000005A000000057
 | |   | |       |_ 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
```

10.3. Motorola S-Record Format

To generate a Motorola S-record output file specify the Control program option **--format=SREC**. Optionally specify the address size with Control program option **--address-size**.

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 k p p c
S00800006C6B707063DD
```

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
```

Chapter 11. 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.

11.1. Structure of a Linker Script File

A script file consists of several definitions. The definitions can appear in any order.

The architecture definition (required)

In essence an *architecture definition* describes how the linker should convert logical addresses into physical addresses for a given type of core. If the core supports multiple address spaces, then for each space the linker must know how to perform this conversion. In this context a physical address is an offset on a given internal or external bus. Additionally the architecture definition contains information about items such as the (hardware) stack and the interrupt vector table.

This specification is normally written by Altium. Altium supplies LSL files in the `include.lsl` directory. The architecture definition of the LSL file should not be changed by you unless you also modify the core's hardware architecture. If the LSL file describes a multi-core system an architecture definition must be available for each different type of core.

See Section 11.4, *Semantics of the Architecture Definition* for detailed descriptions of LSL in the architecture definition.

The derivative definition

The *derivative definition* describes the configuration of the internal (on-chip) bus and memory system. Basically it tells the linker how to convert offsets on the buses specified in the architecture definition into offsets in internal memory. Microcontrollers and DSPs often have internal memory and I/O sub-systems apart from one or more cores. The design of such a chip is called a *derivative*.

Altium provides LSL descriptions of supported derivatives, along with "SFR files", which provide easy access to registers in I/O sub-systems from C and assembly programs. When you build an ASIC or use a derivative that is not (yet) supported by the TASKING tools, you may have to write a derivative definition.

When you want to use multiple cores of the same type, you must instantiate the cores in a derivative definition, since the linker automatically instantiates only a single core for an unused architecture.

See Section 11.5, *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 11.6, *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 11.6.3, *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 11.8, *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
}
```

11.2. 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.

11.2.1. Preprocessing

When the linker loads an LSL file, the linker processes it with a C-style prepocessor. As such, it strips C and C++ comments. You can use the standard ISO C preprocessor directives, such as `#include`, `#define`, `#if/#else/#endif`, `#error`.

For example:

```
#include "arch.lsl"
```

Preprocess and include the file `arch.lsl` at this point in the LSL file.

11.2.2. 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 '`//`'.

11.2.3. 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.

11.2.4. Expressions

The expressions and operators in this section work the same as in ISO C.

```
number           ::= OCT_NUM
                   | DEC_NUM
                   | HEX_NUM

expr             ::= number
                   | symbol_name
                   | unary_op expr
                   | expr binary_op expr
                   | expr ? expr : expr
                   | ( expr )
                   | function_call

unary_op         ::= !    // logical NOT
                   | ~    // bitwise complement
                   | -    // negative value

binary_op        ::= ^    // exclusive OR
                   | *    // multiplication
                   | /    // division
                   | %    // modulus
                   | +    // addition
                   | -    // subtraction
                   | >>   // right shift
                   | <<   // left shift
                   | ==   // equal to
                   | !=   // not equal to
                   | >    // greater than
                   | <    // less than
                   | >=   // greater than or equal to
                   | <=   // less than or equal to
                   | &    // bitwise AND
                   | |    // bitwise OR
                   | &&   // logical AND
                   | ||   // logical OR
```

11.2.5. Built-in Functions

```
function_call    ::= absolute ( expr )
                   | addressof ( addr_id )
                   | exists ( section_name )
                   | max ( expr , expr )
```

```
                          | min ( expr , expr )
                          | sizeof ( size_id )

addr_id                ::= sect : section_name
                          | group : group_name

size_id                ::= sect : section_name
                          | group : group_name
                          | mem : mem_name
```

- Every space, bus, memory, section or group you refer to, must be defined in the LSL file.
- The `addressof()` and `sizeof()` functions with the **`group`** or **`sect`** argument can only be used in the right hand side of an assignment. The `sizeof()` function with the **`mem`** argument can be used anywhere in section layouts.

You can use the following built-in functions in expressions. All functions return a numerical value. This value is a 64-bit signed integer.

absolute()

```
int absolute( expr )
```

Converts the value of *expr* to a positive integer.

```
absolute( "labelA"-"labelB" )
```

addressof()

```
int addressof( addr_id )
```

Returns the address of *addr_id*, which is a named section or group. To get the offset of the section with the name `asect`:

```
addressof( sect: "asect")
```

This function only works in assignments.

exists()

```
int exists( section_name )
```

The function returns 1 if the section *section_name* exists in one or more object file, 0 otherwise. If the section is not present in input object files, but generated from LSL, the result of this function is undefined.

To check whether the section `mysection` exists in one of the object files that is specified to the linker:

```
exists( "mysection" )
```

max()

```
int max( expr, expr )
```

Returns the value of the expression that has the largest value. To get the highest value of two symbols:

```
max( "sym1" , "sym2")
```

min()

```
int min( expr, expr )
```

Returns the value of the expression 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. The **`mem`** argument can be used anywhere in section layouts.

11.2.6. 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.

11.2.7. 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
                     | fill <= fill_values>0|1
                     | write_unit = expr
                     | mapping
```

- A *mem_def* contains exactly one **type** statement.
- A *mem_def* contains exactly one **mau** statement (non-zero size).
- A *mem_def* contains exactly one **size** statement.
- A *mem_def* contains zero or one **speed** statement (if absent, the default speed value is 1).
- A *mem_def* contains zero or one **fill** statement.
- A *mem_def* contains zero or one **write_unit** 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
                          | tag
```

- A *mapping* requires at least the **size** and **dest** statements.
- Each *map_descr* can occur only once.
- You can define multiple mappings from a single source.
- Overlap between source ranges or destination ranges is not allowed.
- If the **src_dbits** or **dest_dbits** statement is not present, its value defaults to the **width** value if the source/destination is a bus, and to the **mau** size otherwise.
- 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.

11.2.8. 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 most one *mapping*.

```
stack_def         ::= stack stack_name ( stack_heap_descr
                            <, stack_heap_descr >>=0 )
```

- A *stack_def* defines a stack with the *stack_name* as a unique name.

```
heap_def          ::= heap heap_name ( stack_heap_descr
                          <, stack_heap_descr >>=0 )
```

- A *heap_def* defines a heap with the *heap_name* as a unique name.

```
stack_heap_descr  ::= min_size = expr
                    | grows = direction
                    | align = expr
                    | fixed
                    | id = expr
                    | tag
```

- The **min_size** statement must be present.
- You can specify at most one **align** statement and one **grows** statement.
- Each stack definition has its own unique **id**, the number specified corresponds to the index in the .CALLS directive as generated by the compiler.

```
direction         ::= low_to_high
                    | high_to_low
```

- If you do not specify the **grows** statement, the stack and heap grow **low-to-high**.

```
copy_table_def    ::= copytable <(  copy_table_descr
                          <, copy_table_descr >>=0 )>0|1
```

- A *space_def* contains at most one **copytable** statement.
- Exactly one copy table must be defined in one of the spaces.

```
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 );

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
```

11.2.9. Derivative Definition

```
derivative_definition
                  ::= derivative derivative_name
                      <( parameter_list )>0|1
                      <extends derivative_name
                              <( argument_list )>0|1 >0|1
                      { <derivative_spec>>=0 }
```

- A *derivative_definition* defines a derivative with the given *derivative_name* as a unique name.

```
derivative_spec   ::= core_def
                     | bus_def
                     | mem_def
                     | section_definition // no processor name
                     | section_setup

core_def          ::= core core_name { <core_descr ;>>=0 }
```

- A *core_def* defines a core with the given *core_name* as a unique name.
- At least one *core_def* must be present in a *derivative_definition*.

```
core_descr        ::= architecture = arch_name
                        <( argument_list )>0|1
                     | endianness = ( endianness_type
                              <, endianness_type>>=0 )
```

- An *arch_name* refers to a defined core architecture.
- Exactly one **architecture** statement must be present in a *core_def*.

11.2.10. 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.

11.2.11. Section Layout Definition and Section Setup

```
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
```

- 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
                    | if_statement
                    | section_creation_statement

group_descr       ::= group <group_name>0|1 <( group_specs )>0|1
                             section_statement
```

- For every group with a name, the linker defines a label.
- No two groups for address spaces of a core can have the same *group_name*.

```
group_specs        ::= group_spec <, group_spec >^{>=0}

group_spec         ::= group_alignment
                     | attributes
                     | copy
                     | nocopy
                     | group_load_address
                     | fill <= fill_values>^{0|1}
                     | group_page
                     | group_run_address
                     | group_type
                     | allow_cross_references
                     | priority = number
                     | tag
```

- The **allow-cross-references** property is only allowed for *overlay* groups.
- Sub groups inherit all properties from a parent group.

```
group_alignment    ::= align = expr

attributes         ::= attributes = <attribute>^{>=1}

attribute          ::= r    // readable sections
                     | w    // writable sections
                     | x    // executable code sections
                     | i    // initialized sections
                     | s    // scratch sections
                     | b    // blanked (cleared) sections

group_load_address
                   ::= load_addr <= load_or_run_addr>^{0|1}

group_page         ::= page <= expr>^{0|1}
                     | page_size = expr <[ range ] <| [ range ]>^{>=0}>^{0|1}

group_run_address  ::= run_addr <= load_or_run_addr>^{0|1}

group_type         ::= clustered
                     | contiguous
                     | ordered
                     | overlay
```

- For *non-contiguous* groups, you can only specify `group_alignment` and `attributes`.
- The **overlay** keyword also sets the **contiguous** property.
- The **clustered** property cannot be set together with **contiguous** or **ordered** on a single group.

```
load_or_run_addr   ::= addr_absolute
                     | addr_range <| addr_range>^{>=0}
```

```
addr_absolute     ::= expr
                    | memory_reference [ expr ]
```

- An absolute address can only be set on *ordered* groups.

```
addr_range        ::= [ expr .. expr ]
                    | memory_reference
                    | memory_reference [ expr .. expr ]
```

- The parent of a group with an `addr_range` or **page** restriction cannot be **ordered**, **contiguous** or **clustered**.
- The end address is not part of the range.

```
memory_reference  ::= mem : <proc_name :>0|1 mem_name </ 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
                    | tag
```

```
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 ;
```

11.3. 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.

11.4. 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
      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
   map

   map
      dest          bus   space
      dest_dbits
      dest_offset
      size
      src_dbits
      src_offset
```

11.4.1. 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
}
```

11.4.2. Defining Internal Buses

With the **bus** keyword you define a bus (the combination of data and corresponding address bus). The bus name is used to identify a bus and does not conflict with other identifiers. Bus descriptions in an architecture definition or derivative definition define *internal* buses. Some internal buses are used to communicate with the components outside the core or processor. Such buses on a processor have physical pins reserved for the number of bits specified with the **width** statements.

- The **mau** field specifies the MAU size (Minimum Addressable Unit) of the data bus. This field is required.
- The **width** field specifies the width (number of address lines) of the data bus. The default value is the MAU size.
- The **map** keyword specifies how this bus maps onto another bus (if so). Mappings are described in Section 11.4.4, *Mappings*.

```
bus bus_name
{
     mau = 8;
     width = 8;
     map ( map_description );
}
```

11.4.3. Defining Address Spaces

With the **space** keyword you define a logical address space. The space name is used to identify the address space and does not conflict with other identifiers.

- The **id** field defines how the addressing space is identified in object files. In general, each address space has a unique ID. The linker locates sections with a certain ID in the address space with the same ID. This field is required.
- The **mau** field specifies the MAU size (Minimum Addressable Unit) of the space. This field is required.
- The **align** value must be a power of two. The linker uses this value to compute the start addresses when sections are concatenated. An align value of *n* means that objects in the address space have to be aligned on *n* MAUs.

- The **page_size** field sets the page alignment and page size in MAUs for the address space. It must be a power of 2. The default value is 1. If one or more page ranges are supplied the supplied value only sets the page alignment. The ranges specify the available space in each page, as offsets to the page start, which is aligned at the page alignment.

  See also the **page** keyword in subsection Locating a group in Section 11.8.2, *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 11.4.4, *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 11.8.3, *Creating or Modifying Special Sections*.

  The stack is described in terms of a minimum size (**min_size**) and the direction in which the stack grows (**grows**). This can be either from **low_to_high** addresses (stack grows upwards, this is the default) or from **high_to_low** addresses (stack grows downwards). The **min_size** is required.

  By default, the linker tries to maximize the size of the stacks and heaps. After locating all sections, the largest remaining gap in the space is used completely for the stacks and heaps. If you specify the keyword **fixed**, you can disable this so-called 'balloon behavior'. The size is also fixed if you used a stack or heap in the software layout definition in a restricted way. For example when you override a stack with another size or select a stack in an ordered group with other sections.

  The **id** keyword matches stack information generated by the compiler with a stack name specified in LSL. This value assigned to this keyword is strongly related to the compiler's output, so users are not supposed to change this configuration.

  Optionally you can specify an alignment for the stack with the argument **align**. This alignment must be equal or larger than the alignment that you specify for the address space itself.

- The **heap** keyword defines a heap in the address space and assigns a name to it. The definition of a heap is similar to the definition of a stack. See also the **heap** keyword in Section 11.8.3, *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 11.8, *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 11.8.2, *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 "vtable" (vector_size=m, size=n, run_addr=x, ...)
```

See the following example of a vector table definition:

```
vector_table "vtable" (vector_size = 4, size = 256, run_addr=0,
                       template=".text.vector_template",
                       template_symbol="_lc_vector_target",
                       vector_prefix=".text.vector.",
                       id_symbol_prefix="foo",
                       no_inline,
                       /* default: empty, or */
                       fill="foo", /* or */
                       fill=[1,2,3,4], /* or */
                       fill=loop)
{
    vector (id=23, fill="main", 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 11.8.3, *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 );
}
```

11.4.4. 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 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:spe:fpi_bus, dest_offset=0x80000000,
                    size=1M);
    map not_cached (dest=bus:spe:fpi_bus, dest_offset=0xa0000000,
                    size=1M, reserved);
}
```

If you are mapping a bus to another bus, the number of data lines of each bus may differ. In this case you have to specify a range of source data lines you want to map (**src_dbits =** *begin..end*) and the range of destination data lines you want to map them to (**dest_dbits =** *first..last*).

- The **src_dbits** argument specifies a range of data lines of the source bus. By default all data lines are mapped.
- The **dest_dbits** argument specifies a range of data lines of the destination bus. By default, all data lines from the source bus are mapped on the data lines of the destination bus (starting with line 0).

From space to space

If you map an address space to another address space (nesting), you can do this by mapping the subspace to the containing larger space. In this example a small space of 64 kB is mapped on a large space of 16 MB.

```
space small
{
   id = 2;
   mau = 4;
   map (src_offset = 0, dest_offset = 0,
        dest = space : large, size = 64k);
}
```

From space to bus

All spaces that are not mapped to another space must map to a bus in the architecture:

```
space large
{
   id = 1;
   mau = 4;
   map (src_offset = 0, dest_offset = 0,
        dest = bus:bus_name, size = 16M );
}
```

From bus to bus

The next example maps an external bus called `e_bus` to an internal bus called `i_bus`. This internal bus resides on a core called `mycore`. The source bus has 16 data lines whereas the destination bus has only 8 data lines. Therefore, the keywords **`src_dbits`** and **`dest_dbits`** specify which source data lines are mapped on which destination data lines.

```
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.

11.5. 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
   fill
   write_unit
   map
section_layout
section_setup

   map
      dest          bus   space
      dest_dbits
      dest_offset
      size
      src_dbits
      src_offset
      reserved
```

11.5.1. Defining a Derivative

With the keyword **derivative** you define a derivative and assign a unique name to it. The name is used to refer to it at other places in the LSL file:

```
derivative name
{
     definitions
}
```

If you are defining multiple derivatives that show great resemblance, you can define the common features in a parent derivative and extend this with a child derivative that contains specific features. The child inherits all features of the parent (cores and memories). With the keyword **extends** you create a child derivative:

```
derivative name_child_deriv extends name_parent_deriv
{
     definitions
}
```

As with a core architecture, a derivative can have any number of parameters. These are identifiers which get values assigned on instantiation or extension of the derivative. You can use them in any expression within the derivative definition.

```
derivative name_child_deriv (parm1,parm2=1)
           extends name_parent_deriv (arguments)
{
     definitions
}
```

11.5.2. 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);
  }
  ```

11.5.3. 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 11.6.3, *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 11.8.2, *Creating and Locating Groups of Sections*).

- The **`mau`** field specifies the MAU size (Minimum Addressable Unit) of the memory. This field is required.

- The **`size`** field specifies the size in MAU of the memory. This field is required.

- The **speed** field specifies a symbolic speed for the memory (1..4): 1 is the slowest, 4 the fastest. The linker uses the relative speed of the memories in such a way, that faster memory is used before slower memory. The default speed is 1.
- The **map** field specifies how this memory maps onto an (internal) bus. The mapping can have a name. Mappings are described in Section 11.4.4, *Mappings*.
- The optional **write_unit** field specifies the minimum write unit (MWU). This is the minimum number of MAUs required in a write action. This is useful to initialize memories that can only be written in units of two or more MAUs. If **write_unit** is not defined the minimum write unit is 0.
- 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;
   write_unit = 4;
   fill = 0xaa;
   size = 64k;
   speed = 2;
   map map_name ( map_description );
}
```

With the **bus** keyword you define a bus in a derivative definition. Buses are described in Section 11.4.2, *Defining Internal Buses*.

11.6. 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
   fill
   write_unit
   map

   map
      dest          bus  space
      dest_dbits
      dest_offset
      size
      src_dbits
      src_offset
      reserved
```

11.6.1. 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
}
```

11.6.2. 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);
}
```

11.6.3. 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;
   write_unit = 4;
   fill = 0xaa;
   size = 64k;
   speed = 2;
   map map_name ( map_description );
}
```

For a description of the keywords, see Section 11.5.3, *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 11.4.2, *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.

11.7. 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
   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
```

11.7.1. Setting up a Section

With the keyword **`section_setup`** you can define stacks, heaps, vector tables, and/or reserved address ranges outside their address space definition.

```
section_setup ::my_space
{
   vector table statements
   reserved address range
   stack definition
   heap definition
}
```

See the subsections Stacks and heaps, Vector table and Reserved address ranges in Section 11.4.3, *Defining Address Spaces* for details on the keywords **`stack`**, **`heap`**, **`vector_table`** and **`reserved`**.

11.8. Semantics of the Section Layout Definition

Keywords in the section layout definition

```
section_layout
   direction     low_to_high  high_to_low
group
   align
   attributes    + -  r w x b i s
   copy
   nocopy
   fill
   ordered
   contiguous
   clustered
   overlay
   allow_cross_references
   load_addr
      mem
   run_addr
      mem
   page
   page_size
   priority
select
stack
   size
heap
   size
reserved
   size
   attributes    r w x
   fill
   alloc_allowed absolute ranged
copytable
section
   size
   blocksize
   attributes    r w x
   fill
   overflow

if
else
```

11.8.1. 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
}
```

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.

11.8.2. 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 11.8.3, *Creating or Modifying Special Sections*.

With the *`group_specifications`* you actually locate the sections in the group. This is described in subsection Locating a group.

Selecting sections for a group

With the keyword **`select`** you can select one or more sections for the group. You can select a section by name or by attributes. If you select a section by name, you can use a wildcard pattern:

*	matches with all section names
?	matches with a single character in the section name
\	takes the next character literally
[abc]	matches with a single 'a', 'b' or 'c' character
[a-z]	matches with any single character in the range 'a' to 'z'

```
group ( ... )
{
   select "mysection";
   select "*";
}
```

The first **`select`** statement selects the section with the name "`mysection`". The second **`select`** statement selects all sections that were not selected yet.

A section is selected by the first select statement that matches, in the union of all section layouts for the address space. Global section layouts are processed in the order in which they appear in the LSL file. Internal core architecture section layouts always take precedence over global section layouts.

- The **`attributes`** field selects all sections that carry (or do not carry) the given attribute. With **+***attribute* you select sections that have the specified attribute set. With **-***attribute* you select sections that do not have the specified attribute set. You can specify one or more of the following attributes:
  - **r** readable sections
  - **w** writable sections
  - **x** executable sections
  - **i** initialized sections
  - **b** sections that should be cleared at program startup
  - **s** scratch sections (not cleared and not initialized)

To select all read-only sections:

```
group ( ... )
{
   select (attributes = +r-w);
}
```

Keep in mind that all section selections are restricted to the address space of the section layout in which this group definition occurs.

- With the **`ref_tree`** field you can select a group of related sections. The relation between sections is often expressed by means of references. By selecting just the 'root' of tree, the complete tree is selected. This is for example useful to locate a group of related sections in special memory (e.g. fast memory). The (referenced) sections must meet the following conditions in order to be selected:

  1. The sections are within the section layout's address space

  2. The sections match the specified attributes

  3. The sections have no absolute restriction (as is the case for all wildcard selections)

  For example, to select the code sections referenced from `foo1`:

  ```
  group refgrp (ordered, contiguous, run_addr=mem:ext_c)
  {
      select ref_tree "foo1" (attributes=+x);
  }
  ```

  If section `foo1` references `foo2` and `foo2` references `foo3`, then all these sections are selected by the selection shown above.

Locating a group

```
group group_name ( group_specifications )
{
   section_statements
}
```

With the *`group_specifications`* you actually define how the linker must locate the group. You can roughly define three things: 1) assign properties to the group like alignment and read/write attributes, 2) define the mutual order in the address space for sections in the group and 3) restrict the possible addresses for the sections in a group.

The linker creates labels that allow you to refer to the begin and end address of a group from within the application software. Labels **`_lc_gb_`***`group_name`* and **`_lc_ge_`***`group_name`* mark the begin and end of the group respectively, where the begin is the lowest address used within this group and the end is the highest address used. Notice that a group not necessarily occupies all memory between begin and end address. The given label refers to where the section is located at run-time (versus load-time).

1. Assign properties to the group like alignment and read/write attributes.

   These properties are assigned to all sections in the group (and subgroups) and override the attributes of the input sections.

   - The **`align`** field tells the linker to align all sections in the group and the group as a whole according to the align value. By default the linker uses the largest alignment constraint of either the input sections or the alignment of the address space.

- The **attributes** field tells the linker to assign one or more attributes to all sections in the group. This overrules the default attributes. By default the linker uses the attributes of the input sections. You can set the **r**, **w**, or **rw** attributes and you can switch between the **b** and **s** attributes.
- The **copy** field tells the linker to locate a read-only section in RAM and generate a ROM copy and a copy action in the copy table. This property makes the sections in the group writable which causes the linker to generate ROM copies for the sections.
- The effect of the **nocopy** field is the opposite of the **copy** field. It prevents the linker from generating ROM copies of the selected sections.

2. Define the mutual order of the sections in the group.

   By default, a group is *unrestricted* which means that the linker has total freedom to place the sections of the group in the address space.

   - The **ordered** keyword tells the linker to locate the sections in the same order in the address space as they appear in the group (but not necessarily adjacent).

     Suppose you have an ordered group that contains the sections 'A', 'B' and 'C'. By default the linker places the sections in the address space like 'A' - 'B' - 'C', where section 'A' gets the lowest possible address. With **direction=high_to_low** in the **section_layout** space properties, the linker places the sections in the address space like 'C' - 'B' - 'A', where section 'A' gets the highest possible address.

   - The **contiguous** keyword tells the linker to locate the sections in the group in a single address range. Within a contiguous group the input sections are located in arbitrary order, however the group occupies one contiguous range of memory. Due to alignment of sections there can be 'alignment gaps' between the sections.

     When you define a group that is both **ordered** and **contiguous**, this is called a *sequential* group. In a sequential group the linker places sections in the same order in the address space as they appear in the group and it occupies a contiguous range of memory.

   - The **clustered** keyword tells the linker to locate the sections in the group in a number of *contiguous* blocks. It tries to keep the number of these blocks to a minimum. If enough memory is available, the group will be located as if it was specified as **contiguous**. Otherwise, it gets split into two or more blocks.

     If a contiguous or clustered group contains *alignment gaps*, the linker can locate sections that are not part of the group in these gaps. To prevent this, you can use the **fill** keyword. If the group is located in RAM, the gaps are treated as reserved (scratch) space. If the group is located in ROM, the alignment gaps are filled with zeros by default. You can however change the fill pattern by specifying a bit pattern. The result of the expression, or list of expressions, is used as values to write to memory, each in MAU.

   - The **overlay** keyword tells the linker to overlay the sections in the group. The linker places all sections in the address space using a contiguous range of addresses. (Thus an overlay group is automatically also a contiguous group.) To overlay the sections, all sections in the overlay group share the same run-time address.

For each input section within the overlay the linker automatically defines two symbols. The symbol **_lc_cb_***section_name* is defined as the load-time start address of the section. The symbol **_lc_ce_***section_name* is defined as the load-time end address of the section. C (or assembly) code may be used to copy the overlaid sections.

If sections in the overlay group contain references between groups, the linker reports an error. The keyword **allow_cross_references** tells the linker to accept cross-references. Normally, it does not make sense to have references between sections that are overlaid.

```
group ovl (overlay)
{
    group a
    {
        select "my_ovl_p1";
        select "my_ovl_p2";
    }
    group b
    {
        select "my_ovl_q1";
    }
}
```

It may be possible that one of the sections in the overlay group already has been defined in another group where it received a load-time address. In this case the linker does not overrule this load-time address and excludes the section from the overlay group.

3. Restrict the possible addresses for the sections in a group.

   The load-time address specifies where the group's elements are loaded in memory at download time. The run-time address specifies where sections are located at run-time, that is when the program is executing. If you do not explicitly restrict the address in the LSL file, the linker assigns addresses to the sections based on the restrictions relative to other sections in the LSL file and section alignments. The program is responsible for copying overlay sections at appropriate moment from its load-time location to its run-time location (this is typically done by the startup code).

   - The **run_addr** keyword defines the run-time address. If the run-time location of a group is set explicitly, the given order between groups specify whether the run-time address propagates to the parent group or not. The location of the sections a group can be restricted either to a single absolute address, or to a number of address ranges (not including the end address). With an expression you can specify that the group should be located at the absolute address specified by the expression:

     ```
     group (run_addr = 0xa00f0000)
     ```

     You can use the '**[***offset***]**' variant to locate the group at the given absolute offset in memory:

     ```
     group (run_addr = mem:A[0x1000])
     ```

     A range can be an absolute space address range, written as **[** *expr .. expr* **]**, a complete memory device, written as **mem:***mem_name*, or a memory address range, **mem:**mem_name**[***expr .. expr* **]**

```
group (run_addr = mem:my_dram)
```

You can use the '**|**' to specify an address range of more than one physical memory device:

```
group (run_addr = mem:A | mem:B)
```

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 11.4.3, *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";
  }
  ```

11.8.3. 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 11.4.3, *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.

  ```
  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

Properties set in LSL		**Resulting section properties**		
attributes	**filled**	**access**	**memory**	**content**
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** and **load_addr** attributes.

  The **fill** field contains a bit pattern that the linker writes to all unused space in the output section. When all input sections have an image (code/data) you must specify a fill pattern. If you do not specify a fill pattern, all input sections must be scratch sections. The fill pattern is aligned at the start of the output section.

  As with a reserved section you can use the **attributes** field to set the access type of the output section.

```
group ( ... )
{
   section "myoutput" ( size = 4k, attributes = rw,
                        fill = 0xaa )
   {
      select "myinput1";
      select "myinput2";
   }
}
```

  The available room for input sections is determined by the **size**, **blocksize** and **overflow** fields. With the keyword **size** you specify the fixed size of the output section. Input sections are placed from output section start towards higher addresses (offsets). When the end of the output section is reached and one or more input sections are not yet placed, an error is emitted. If however, the **overflow** field is set to another output section, remaining sections are located as if they were selected for the overflow output section.

```
group ( ... )
{
  section "tsk1_data" (size=4k, attributes=rw, fill=0,
                       overflow = "overflow_data")
  {
          select ".data.tsk1.*"
  }
  section "tsk2_data" (size=4k, attributes=rw, fill=0,
                       overflow = "overflow_data")
  {
          select ".data.tsk2.*"
  }
  section "overflow_data" (size=4k, attributes=rx,
                           fill=0)
  {
  }
}
```

With the keyword **`blocksize`** , the size of the output section will adapt to the size of its content. For example:

```
group flash_area (run_addr = 0x10000)
{
   section "flash_code" (blocksize=4k, attributes=rx,
                         fill=0)
   {
     select "*.flash";
   }
}
```

If the content of the section is 1 mau, the size will be 4 kB, if the content is 11 kB, the section will be 12 kB, etc. If you use **`size`** in combination with **`blocksize`**, the **`size`** value is used as default (minimal) size for this section. If it is omitted, the default size will be of **`blocksize`**. It is not allowed to omit both **`size`** and **`blocksize`** from the section definition.

The linker creates two labels to mark the begin and end of the section, **`_lc_ub_`***name* for the begin of the section and **`_lc_ue_`***name* for the end of the output section.

Copy table

- The keyword **`copytable`** tells the linker to select a section that is used as *copy table*. The content of the copy table is created by the linker. It contains the start address and length of all sections that should be initialized by the startup code.

  The linker creates two labels to mark the begin and end of the section, **`_lc_ub_table`** for the begin of the section and **`_lc_ue_table`** for the end of the copy table. The linker generates a copy table when a reference to either of the section labels exists in one of the input object files.

11.8.4. 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_cp" := "_lc_ub_table";
    // when the symbol _lc_cp occurs as an undefined reference
    // in an object file, the linker generates a copy table
}
```

11.8.5. 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 12. 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 VX-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.

12.1. 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.

12.2. Declarations and Initialization (DCL)

DCL30-C Declare objects with appropriate storage durations

The lifetime of an automatic object ends when the function returns, which means that a pointer to the object becomes invalid.

DCL31-C Declare identifiers before using them

The ISO C90 standard allows implicit typing of variables and functions. Because implicit declarations lead to less stringent type checking, they can often introduce unexpected and erroneous behavior or even security vulnerabilities. The ISO C99 standard requires type identifiers and forbids implicit function declarations. For backwards compatibility reasons, the VX-toolset C compiler assumes an implicit declaration and continues translation after issuing a warning message (W505 or W535).

DCL32-C Guarantee that mutually visible identifiers are unique

The compiler encountered two or more identifiers that are identical in the first 31 characters. The ISO C99 standard allows a compiler to ignore characters past the first 31 in an identifier. Two distinct identifiers that are identical in the first 31 characters may lead to problems when the code is ported to a different compiler.

DCL35-C Do not invoke a function using a type that does not match the function definition

This warning is generated when a function pointer is set to refer to a function of an incompatible type. Calling this function through the function pointer will result in undefined behavior. Example:

```
void my_function(int a);
int main(void)
{
  int (*new_function)(int a) = my_function;
  return (*new_function)(10); /* the behavior is undefined */
}
```

12.3. Expressions (EXP)

EXP01-C Do not take the size of a pointer to determine the size of the pointed-to type

The size of the object(s) allocated by malloc(), calloc() or realloc() should be a multiple of the size of the base type of the result pointer. Therefore, the sizeof expression should be applied to this base type, and not to the pointer type.

EXP12-C Do not ignore values returned by functions

The compiler gives this warning when the result of a function call is ignored at some place, although it is not ignored for other calls to this function. This warning will not be issued when the function result is ignored for all calls, or when the result is explicitly ignored with a (void) cast.

EXP30-C Do not depend on order of evaluation between sequence points

Between two sequence points, an object should only be modified once. Otherwise the behavior is undefined.

EXP32-C Do not access a volatile object through a non-volatile reference

If an attempt is made to refer to an object defined with a volatile-qualified type through use of an lvalue with non-volatile-qualified type, the behavior is undefined.

EXP33-C Do not reference uninitialized memory

Uninitialized automatic variables default to whichever value is currently stored on the stack or in the register allocated for the variable. Consequently, uninitialized memory can cause a program to behave in an unpredictable or unplanned manner and may provide an avenue for attack.

EXP34-C Ensure a null pointer is not dereferenced

Attempting to dereference a null pointer results in undefined behavior, typically abnormal program termination.

EXP37-C Call functions with the arguments intended by the API

When a function is properly declared with function prototype information, an incorrect call will be flagged by the compiler. When there is no prototype information available at the call, the compiler cannot check the number of arguments and the types of the arguments. This message is issued to warn about this situation.

EXP38-C Do not call offsetof() on bit-field members or invalid types

The behavior of the offsetof() macro is undefined when the member designator parameter designates a bit-field.

12.4. 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.

12.5. 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.

12.6. 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.

12.7. 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.

12.8. 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.

12.9. 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.

12.10. 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.

12.11. 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 13. MISRA-C Rules

This chapter contains an overview of the supported and unsupported MISRA C rules.

13.1. MISRA-C:1998

This section lists all supported and unsupported MISRA-C:1998 rules.

See also *Section 3.8.2, 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.

13.2. MISRA-C:2004

This section lists all supported and unsupported MISRA-C:2004 rules.

See also Section 3.8.2, *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)	`typedefs` that indicate size and signedness should be used in place of the basic types.
6.4	(R)	Bit-fields shall only be defined to be of type `unsigned int` or `signed int`.
6.5	(R)	Bit-fields of type `signed int` shall be at least 2 bits long.

Constants

7.1	(R)	Octal constants (other than zero) and octal escape sequences shall not be used.

Declarations and definitions

8.1	(R)	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.
8.2	(R)	Whenever an object or function is declared or defined, its type shall be explicitly stated.
8.3	(R)	For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.
8.4	(R)	If objects or functions are declared more than once their types shall be compatible.
8.5	(R)	There shall be no definitions of objects or functions in a header file.
8.6	(R)	Functions shall be declared at file scope.
8.7	(R)	Objects shall be defined at block scope if they are only accessed from within a single function.
8.8	(R)	An external object or function shall be declared in one and only one file.

	8.9	(R)	An identifier with external linkage shall have exactly one external definition.
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.

	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.