TASKING VX-toolset for C166 User Guide

TASKING VX-toolset for C166 User Guide

Table of Contents

Chapter 1. C Language

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

The TASKING C compiler for C166® fully supports the ISO-C standard and add 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

Fundamental data types

The C compiler supports the ISO C99 defined data types. The sizes of these types are shown in the following table.

C Type	Size	Align	Limits
__bit	1	1	0 or 1
_Bool	8	8	0 or 1
signed char	8	8	[-0x80, +0x7F]
unsigned char	8	8	[0, 0xFF]
short	16	16	[-0x8000, +0x7FFF]
unsigned short	16	16	[0, 0xFFFF]
int	16	16	[-0x8000, +0x7FFF]

C Type	Size	Align	Limits
unsigned int	16	16	[0,0xFFFF]
enum *	8 16	8 16	[-0x80, +0x7F] or [0, 0xFF] [-0x8000, +0x7FFF] or [0,0xFFFF]
long	32	16	[-0x80000000, +0x7FFFFFFF]
unsigned long	32	16	[0,0xFFFFFFFF]
long long	64	16	[-0x8000000000000000, +0x7FFFFFFFFFFFFFFF]
unsigned long long	64	16	[0, 0xFFFFFFFFFFFFFFFF]
float (23-bit mantissa)	32	16	[−3.402E+38, −1.175E-38] [+1.175E-38, +3.402E+38]
double long double (52-bit mantissa)	64	16	[-1.797E+308, -2.225E-308] [+2.225E-308, +1.797E+308]
_Imaginary float	32	16	[−3.402E+38i, −1.175E-38i] [+1.175E-38i, +3.402E+38i]
_Imaginary double _Imaginary long double	64	16	[-1.797E+308i, -2.225E-308i] [+2.225E-308i, +1.797E+308i]
_Complex float	64	16	real part + imaginary part
_Complex double _Complex long double	128	16	real part + imaginary part
__near pointer to data or function	16	16	[0,0xFFFF]
__far pointer **	32	16	[0,0xFFFFFF]
__shuge pointer **	32	16	[0,0xFFFFFF]
__huge pointer to data or function	32	16	[0,0xFFFFFF]

* When you use the `enum` type, the compiler will use the smallest sufficient type (`char`, `unsigned char` or `int`), unless you use C compiler option **--integer-enumeration** (always use 16-bit integers for enumeration).

** __far pointers are calculated using 14-bit arithmetic, __shuge pointers are calculated using 16-bit arithmetic.

Automatic bit objects never reside on the user stack, because the stack is not bit-addressable. So, it is not possible to take the address of an automatic bit object, or to create automatic bit-arrays, because these operations would force an object on the stack.

Aggregate and union types

Aggregate types are aligned on 16 bits by default. All members of the aggregate types are aligned as required by their individual types as listed in the table above. The struct/union data types may contain bit-fields. The allowed bit-field fundamental data types are `_Bool`, `(un)signed char` and `(un)signed int`. The maximum bit-field size is equal to that of the type's size. For the bit-field types the same rules

regarding to alignment and signed-ness apply as specified for the fundamental data types. In addition, the following rules apply:

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

- A bit-field of a particular type cannot cross a boundary as is specified by its maximum width. For example, a bit-field of type `short` cannot cross a 16-bit boundary.

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

- An unnamed bit-field creates a gap that has the size of the specified width.

- As a special case, an unnamed bit-field having width 0 (zero) prevents any further bit-field from residing in the storage unit corresponding to the type of the zero-width bit-field.

Bit structures

The `__bit` data type is allowed as a struct/union member, with the restriction that no other type than `__bit` is member of this structure. This creates a bit-structure that is allocated in bit-addressable memory. Its alignment is 1 bit.

There are a number of restrictions to bit-structures. They are described below:

- It is not possible to pass a bit-structure argument to a function.

- It is not possible to return a bit-structure.

- It is not possible to make an automatic bit-structure.

The reason for these restrictions is that a bit-structure must be allocated in bit-addressable memory, which the user stack is not.

__bitsizeof() operator

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

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

1.2. Changing the Alignment: __unaligned and __packed__

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

Suppress alignment

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

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

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

Example:

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

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

Packed structures

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

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

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

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

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

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

1.3. Accessing Memory

The TASKING VX-toolset for C166 internally knows the following address types:

• 32-bit linear, 'huge' addresses. The address notation is in bytes, starts at 0 and ends at 16M.

- 32-bit paged, 'far' addresses. In the address notation the high word contains the 10-bit page number and the low word contains the 14-bit offset within the 16 kB page.

- 16-bit, 'near' addresses. The high 2 bits contain the DPP number and the low 14 bits are the offset within the 16 kB page.

- 12-bit bit-addressable addresses. This embodies an 8-bit word offset in the bit-addressable space and a 4-bit bit number.

- 8-bit SFR addresses. This is an offset within the SFR space or within the extended SFR space.

The TASKING VX-toolset for C166 toolset has several keywords you can use in your C source to specify memory locations. This is explained in the sub-sections that follow.

1.3.1. Memory Type Qualifiers

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

You can specify the following memory types:

Qualifier	Description	Location	Maximum object size	Pointer size	Pointer arithmetic	Section name and type [*]
__bit [**]	Bit addressable	Bit addressable memory	1 bit	16-bit	12-bit	bit
__bita	Bit addressable	Bit addressable memory	Size of bit addressable memory	16-bit	16-bit	bita
__iram	Internal RAM data	Internal RAM	Size of internal RAM	16-bit	16-bit	iram
__near	Near data	In the 4 near data pages	16 kB	16-bit	16-bit	near
__far	Far data	Anywhere	16 kB	32-bit	14-bit	far
__shuge	Segmented huge data	Anywhere	64 kB	32-bit	16-bit	shuge
__huge	Huge data	Anywhere	no limit	32-bit	32-bit	huge

[*] The default section name is equal to a combination of the generated section type and the object name. You can change the section name with the `#pragma section` or command line option **--rename-sections**.

[**] `__bit` is not a real qualifier, it is in fact a data type with an implicit memory type of type bit.

There are no SFR qualifiers. SFRs are accessible in the near address space. The compiler knows which absolute address ranges belong to SFR areas and extended SFR areas and knows which addresses are bit addressable. The compiler generates the appropriate SFR addressing modes for these addresses.

Examples using explicit memory types

```
__bita    unsigned char  bitbyte;
__bita    unsigned short bitword;
__near    char  text[] = "No smoking";
__far     int   array[10][4];
```

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:

```
unsigned char   __bita    bitbyte;
unsigned short  __bita    bitword;
char    __near   text[] = "No smoking";
int     __far    array[10][4];
```

__far and __shuge code generation

The `__far` and `__shuge` qualifiers have only very little difference in code generation. There are two basic differences:

- Accessing `__far` objects is done using EXTP instructions and accessing `__shuge` objects is done using EXTS instructions. This has no impact on code size or execution speed, and therefore it is in general preferred to use `__shuge`, because objects can be as large as 64 kB, while with `__far` the size of a single object is limited to 16 kB.

- Code generation for accessing objects on stack is a little bit more efficient for `__far` pointers than for `__shuge` pointers.

1.3.1.1. Pointers with Memory Type Qualifiers

Pointers for the C166 can have two types: a 'logical' type and a memory type. For example,

```
char __far * __near p;
```

means `p` has memory type `__near` (`p` itself is allocated in near data), but has logical type 'character in target memory space far'. The memory type qualifier used to the left of the '*', specifies the target memory of the pointer, the memory type qualifier used to the right of the '*', specifies the storage memory of the pointer.

__far and __shuge pointer comparison

By default all `__far` pointer arithmetic is 14-bit. This implies that comparison of `__far` pointers is also done in 14-bit. For `__shuge` the same is true, but then with 16-bit arithmetic. This saves code significantly, but has the following implications:

- Comparing pointers to different objects is not reliable. It is only reliable when it is known that these objects are located in the same page.

- Comparing with NULL is not reliable. Objects that are located in another page at offset 0x0000 have the low 14 bits (the page offset) zero and will also be evaluated as NULL. In the following example the if(p) is false, because the page offset of p is zero:

```
__far int   i __at(0x10000);
__far int * p = &i;
if( p ) p++;
```

In most cases these restrictions will not yield any problems, but in case problems exist, the following solutions are available:

- Cast the problematic comparison to long, e.g.: if((long)p)

- Use the C compiler option **-AF** to tell the compiler to generate 32-bit pointer comparisons. Note that it is also required to rebuild the C library if C library routines are used.

Pointer conversions

Conversions of pointers with the same qualifiers are always allowed. The following table contains the additionally allowed pointer conversions. Other pointer conversions are not allowed to avoid possible run-time errors.

Source pointer	Destination pointer
__bita	__iram
__bita	__near
__bita	__far
__bita	__shuge
__bita	__huge
__iram	__near
__iram	__far
__iram	__shuge
__iram	__huge
__near	__far
__near	__shuge
__near	__huge
__far	__shuge
__far	__huge
__shuge	__huge

__near, __bita, __iram (16-bit) pointer conversions to and from non-pointer types:

- A conversion from a 32-bit integer to a 16-bit pointer, or from a 16-bit pointer to a 32-bit integer, is implemented as a 32-bit linear address conversion.

- All other non-pointer conversions to and from a 16-bit pointer are implemented as a conversion to or from a 16-bit integer type.

`__far` (32-bit) pointer conversions to and from non-pointer types:

- A conversion from a 16-bit integer to a `__far` pointer, or from a `__far` pointer to a 16-bit integer, is implemented as a 16-bit linear address conversion. The behavior of a `__far` pointer to 16-bit integer conversion is undefined when `__far` pointer contains an address with page number larger than 3.

- A conversion from a 32-bit integer to a `__far` pointer, or from a `__far` pointer to a 32-bit integer, is implemented as a 32-bit linear address conversion.

- All other non-pointer conversions to and from a `__far` pointer are implemented as a conversion to or from a 32-bit integer type.

`__(s)huge` (32-bit) pointer conversions to and from non-pointer types:

- All non-pointer conversions to and from a `__(s)huge` pointer are implemented as a conversion to or from a 32-bit integer.

1.3.2. Memory Models

The C compiler supports four data memory models, listed in the following table.

Memory model	Letter	Default data memory type
Near	n	__near
Far	f	__far
Segmented Huge	s	__shuge
Huge	h	__huge

Each memory model defines a default memory type for objects that do not have a memory type qualifier specified. By default, the C166 compiler uses the near memory model. With this memory model the most efficient code is generated. With the C compiler option **--model** you can specify another memory model.

For information on the memory types, see Section 1.3.1, *Memory Type Qualifiers*.

__MODEL__

The compiler defines the preprocessor symbol __MODEL__ to the letter representing the selected memory model. This can be very helpful in making conditional C code in one source module, used for different applications in different memory models.

Example:

```
#if __MODEL__ == 'f'
/* this part is only for the far memory model */
...
#endif
```

DPP usage

The compiler uses EXTP/EXTS instructions to access far, shuge and huge data in all data models. This means that it does not use DPP loads and DPP prefixes. All DPPs point to the near data space at anytime.

The advantages of not using DPPs are:

• There are always four near data pages.

• Interrupt functions will not save/restore any DPPs.

• You can use a DPP for your own purpose by letting the linker not assign the DPP to a near page. The best way to do this is to assign the DPP in LSL to an unused page in memory and reserve that page.

• Bit 14 and bit 15 do not need to be masked when converting a pointer to stack (which is near) to far.

Near data

Near data is paged in all memory models. The linker takes care of assigning DPPs in the code.

With a trick in the LSL file (by defining the __CONTIGUOUS_NEAR macro) it is possible to remove this page restriction and get a linear space, even if the near data pages are scattered throughout the memory. The linker takes care of locating the sections in such a way that the compiler can assume them to be contiguous through the near data pages. This also implies that the linker can split sections and put parts in non-consecutive near data pages. When this LSL trick is applied, you should be very cautious when accessing near data with far or shuge pointers, because objects may cross page or segment boundaries.

Stack

In all memory models the stack is restricted to 16 kB and must be in-page. With a trick in the LSL file (by defining the __CONTIGUOUS_NEAR macro) it is possible to remove the page limitation of the stack. But this should only be done when you do not use far, shuge or huge pointers to access objects on the stack, because page or segment boundaries may be crossed, and the compiler will use the begin of stack to perform casts to stack objects.

For XC16x and Super10 derivatives, multiple stacks are created in the LSL file, one for each local register bank. The C startup code controls the creation of these stacks, by referring the begin of stack symbols.

Heap

In the far, huge and segmented huge models the heap is located as huge data. The memory allocation routines in the C library will take care of keeping the data in pages or segments for far and shuge data. In the near data model the default heap is located as near data. Optionally a huge heap can be allocated allocating far/shuge/huge data.

Threshold

In the far, segmented huge and huge data models the compiler supports a threshold for allocating default objects in near data. Objects that are smaller than or equal to the threshold area automatically allocated in near data. The threshold can be defined on the command line with option **--near-threshold**. By default the threshold is 0 (off), which means that all data is allocated in the default memory space. In the far, huge and segmented huge memory models, near data sections that result from the threshold optimization

will be marked to be located inpage, because sections may not cross page boundaries when access through an external far, huge or shuge declaration is done.

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

Just like you can declare a variable in a specific part of memory (using memory type 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 (huge) address. If you use this keyword on `__bit` objects or functions, the address is a bit address.

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

Examples

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

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

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

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

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

The function `f` is placed at address 0xf100.

Restrictions

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

* The argument of the `__at()` attribute must be a constant address expression.

* You can place only global variables at absolute addresses. Parameters of functions, or automatic variables within functions cannot be placed at absolute addresses.

* A variable that is declared `extern`, is not allocated by the compiler in the current module. Hence it is not possible to use the keyword `__at()` on an external variable. Use `__at()` at the definition of the variable.

* You cannot place structure members at an absolute address.

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

1.3.4. Accessing Bits

There are several methods to access single bits in the bit-addressable area. The compiler generates efficient bit operations where possible.

Masking and shifting

The classic method to extract a single bit in C is masking and shifting.

```
__bita unsigned short bitword;
void foo( void )
{
    if( bitword & 0x0004 )    // bit 2 set?
    {
        bitword &= ~0x0004;   // clear bit 2
    }
    bitword |= 0x0001;        // set bit 0;
}
```

Built-in macros __getbit() and __putbit()

The compiler has the built-in macros `__getbit()` and `__putbit()`. These macros expand to shift/and/or combinations to perform the required result.

```
__bita unsigned short bw;
void foo( void )
{
    if( __getbit( bw, 2 ) )
    {
        __putbit( 0, bw, 2 );
    }
    __putbit( 1, bw, 0 );
}
```

Accessing bits using a struct/union combination

```
typedef __bita union
{
    unsigned short word;
    struct
    {
        int  b0 : 1;
        int  b1 : 1;
        int  b2 : 1;
        int  b3 : 1;
        int  b4 : 1;
        int  b5 : 1;
        int  b6 : 1;
        int  b7 : 1;
        int  b8 : 1;
        int  b9 : 1;
        int  b10: 1;
        int  b11: 1;
        int  b12: 1;
        int  b13: 1;
        int  b14: 1;
```

```
        int  b15: 1;
    } bits;
} bitword_t;

bitword_t bw;

void foo( void )
{
    if( bw.bits.b3 )
    {
        bw.bits.b3 = 0;
    }
    bw.bits.b0 = 1;
}

void reset( void )
{
    bw.word = 0;
}
```

Declaring a bit variable with __atbit() (backwards compatibility only)

For backwards compatibility, you can still use the `__atbit()` keyword to define a bit symbol as an alias for a single bit in a bit-addressable object. However, we recommend that you use one of the methods described above to access a bit.

The syntax of `__atbit()` is:

`__atbit(object,offset)`

where *object* is a bit-addressable object and *offset* is the bit position in the object.

The following restrictions apply:

* This keyword can only be applied to `__bit` type symbols.

* The bit must be defined `volatile` explicitly. The compiler issues a warning if the bit is not defined volatile and makes the bit volatile.

* The bitword can be any `volatile` bit-addressable (`__bita`) object. The compiler issues a warning if the bit-addressable object was not volatile and makes it volatile.

* The bit symbol cannot be used as a global symbol. An extern on the bit variable, without `__atbit()`, will lead to an unresolved external message from the linker, so therefore `__atbit()` is required.

Examples

```
/* Module 1 */
volatile __bita unsigned short bitword;
volatile __bit b __atbit( bitword, 3 );

/* Module 2 */
```

```
extern volatile __bita unsigned short bitword;
extern volatile __bit b __atbit( bitword, 3);
```

Drawbacks of __atbit()

- `__atbit()` requires all involved objects to be volatile. If your application does not require these objects to be volatile, you may see in many cases that the generated code is less optimal than when the objects were not volatile. The reason for that is that the compiler must generate each read and write access for volatile objects as written down in the C code. Fortunately the standard C language provides methods to achieve the same result as with `__atbit()`. The compiler is smart enough to generate efficient bit operations where possible.

- No debug information is generated for objects defined with `__atbit()`, meaning that these objects are not visible in the debugger.

1.3.5. Accessing Hardware from C

Using Special Function Registers

It is easy to access Special Function Registers (SFRs) that relate to peripherals from C. The SFRs are defined in a special function register file (`*.sfr` and `*.asfr`) as symbol names for use with the compiler. An SFR file contains the names of the SFRs and the bits in the SFRs. These SFR files are also used by the assembler and the simulator engine. The debugger and integrated environment use the XML variants of the SFR files. The XML files include full descriptions of the SFRs and the bit-fields. Also the bit-field values are described. To decrease compile time the `.sfr` and `.asfr` files do not contain the descriptions. The `.sfr` and `.asfr` files are in written C and are derived from the XML files.

SFRs in the SFR area and extended SFR area are addressed in the near address space. The compiler knows the effective address ranges and generates SFR addressing modes for this. The generated addressing modes to access the registers depend on the address. Some SFRs cannot be addressed with a REG addressing mode, although they are within the SFR area or the extended SFR area. These registers are:

RSTCON	0xF1E0
RSTCON2	0xF1E2
SYSSTAT	0xF1E4

The compiler will never emit REG addressing for these addresses.

You can find a list of defined SFRs and defined bits by inspecting the SFR file for a specific processor. The files are named reg*cpu*`.sfr` and reg*cpu*`.asfr`, where *cpu* is the CPU specified with the C compiler option **--cpu**. The compiler automatically includes this register file, unless you specify option **--no-tasking-sfr**. The files are located in the `sfr` subdirectory of the standard `include` directory.

For new devices (XC2xxx, XE16x) most SFR names use a standard naming convention prescribed by Infineon:

- *UNIT_SFRNAME* for SFRs

- *UNIT_SFRNAME_BITNAME* for bit-fields, when using the standard `.sfr` files

For example:

```
IMB_MAR0
IMB_MAR0_HREAD0
```

For some of these names aliases are defined that do not include the unit name. For example, SYSCON0 as an alias for SCU_SYSCON0. As a rule of thumb aliases are available only where needed to make the cstart.c code generic between older and newer devices.

.sfr - the standard SFR file format

These files are read by the C++ compiler, C compiler, assembler and simulator engine. The SFRs and SFR bit-fields are defined as C preprocessor macros using #define-s. The *.sfr files are the default for the toolset.

Example use in C (with use of alias definitions):

```c
void set_sfr(void)
{
    P0L = 0x88;    // use port name
    AD3 = 1;       // use of bit name
    if (AD4 == 1)
    {
      AD3 = 0;
    }
    IEN = 1;       // use of bit name
}
```

The compiler generates (with option **--cpu=c167**):

```
    movw 0xff00,#0x88
    bset 0xff00.3
    jnb  0xff00.4,_2
    bclr 0xff00.3
_2:
    bset 0xff10.11
```

When the SFR contains fields, the layout of the SFR is defined by a typedef-ed union of a struct with bits, a signed integer and an unsigned integer:

```c
typedef volatile union __PSW_union

{
    struct __PSW_struct
    {
        unsigned n              : 1;
        unsigned c              : 1;
        unsigned v              : 1;
        unsigned z              : 1;
        unsigned e              : 1;
        unsigned mulip          : 1;
        unsigned usr0           : 1;
```

```
        unsigned usr1           : 1;
        unsigned bank           : 2;
        unsigned hlden          : 1;
        unsigned ien            : 1;
        unsigned ilvl           : 4;
    } B;
    int I;
    unsigned U;
} __PSW_type;
```

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

Note that the bit-field names are in lower case to avoid conflicts with their macro definition.

The base SFR is defined by a cast to a 'typedef-ed union' pointer. The SFR address is given in parenthesis. This definition is the same in the `*.sfr` and `*.asfr` files and is therefore also used in the C startup code (`cstart.c`)

```
#define __PSW (*( __PSW_type *)0xFF10)
```

The definition of the actual SFR name as it should be used in your code:

```
#define PSW      __PSW.U
```

A bit-field is defined as:

```
#define PSW_N    __PSW.B.n
```

You can also use an alias definition for the bit-field:

```
#define N        PSW_N
```

Note that if the bit-field in the `struct` definition would be uppercase it would give a name clash.

.asfr - the alternative SFR file format

These files can optionally be used by the C compiler instead of the `.sfr` files. These files are named reg*cpu*.asfr. This format does not contain alias definitions for SFRs and SFR bit-fields.

You can select `.asfr` instead of `.sfr` files, with option **--alternative-sfr-file**.

Example in C when you use this alternative format:

```
void set_sfr(void)
{
    P0L.U = 0x88;       // use port name as unsigned integer
    P0L.I = 136;        // use port name as signed integer
    P0L.B.AD3 = 1;      // use of bit name
    if (P0L.B.AD4 == 1)
    {
      P0L.B.AD3 = 0;
    }
```

```
    PSW.B.IEN = 1;        // use of bit name
}
```

The compiler generates (with options **--cpu=c167 --alternative-sfr-file**):

```
    movw 0xff00,#0x88
    movw 0xff00,#0x88
    bset 0xff00.3
    jnb  0xff00.4,_2
    bclr 0xff00.3
_2:
    bset 0xff10.11
```

When the SFR contains fields, the layout of the SFR is defined by a typedef-ed union of a struct with bits, a signed integer and an unsigned integer:

```
typedef volatile union  __PSW_union
{
    struct __PSW_struct
    {
        unsigned N               : 1;
        unsigned C               : 1;
        unsigned V               : 1;
        unsigned Z               : 1;
        unsigned E               : 1;
        unsigned MULIP           : 1;
        unsigned USR0            : 1;
        unsigned USR1            : 1;
        unsigned BANK            : 2;
        unsigned HLDEN           : 1;
        unsigned IEN             : 1;
        unsigned ILVL            : 4;
    } B;
    int I;
    unsigned U;
} __PSW_type;
```

Note that the bit-fields are in upper case.

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

The base SFR is defined by a cast to a 'typedef-ed union' pointer. The SFR address is given in parenthesis. This definition is the same in the `*.sfr` and `*.asfr` files and is therefore also used in the C startup code (`cstart.c`)

```
#define __PSW (*( __PSW_type *)0xFF10)
```

The definition of the actual SFR name as it should be used in your code:

```
#define PSW        __PSW
```

To access the SFRs in the alternative format, you should use the following syntax:

```
.U    Full SFR, unsigned
.I    Full SFR, signed
.B.   Bit-field
```

For example:

```
WDTREL.U = 0x300;
U0C0_IN00.I = -12;
PSW.B.IEN = 1;
```

Choosing between the standard and alternative SFR file format

It depends on the coding of the application which format can be used. It is recommended to use the alternative SFR file format for new applications because of the following benefits:

• Less namespace pollution because of the lack of alias definitions.

  In the standard SFR file format (`.sfr`) all bit-fields are defined as C preprocessor macros using `#define`. This gives namespace pollution with the risk of a name clash with the application. For example, applications that define a macro N are not so seldom, and will give a clash with the bit-field N.

• Faster compilation due to smaller SFR files.

  The compiler needs significantly more time to process all the additional bit-field definitions in a standard SFR file (`.sfr`). Processing a `.asfr` file may be twice as fast as processing a `.sfr` file. For new devices with a large amount of SFRs this will result in measurable shorter build times.

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

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

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

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*.
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. You can use byte registers RL0 - RL7, RH0 - RH7 and word registers R0 - R15. Note that saving too much 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
b	byte register	RL0 - RL7, RH0 - RH7	input/output constraint
w	word register	R0 - R15	input/output constraint
i	indirect address register	R0 - R3	input constraint only
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() statement 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 b; 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 assign the result to the output variable.

```
char out;
void addone( void )
{
    __asm( "MOV %0,#1"
            : "=b" (out) );
}
```

Generated assembly code:

```
MOV rh4,#1
movb _out,rh4
```

Example 3: using input parameters

Assign a variable to an SFR. A word register is chosen for the parameter because of the constraint w; 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 initsfr( void )
{
    __asm( "MOVW  P0L,%0"
            :
```

```
        : "w" (in) );
}
```

Generated assembly code:

```
    movw  r11, _in
    MOVW  P0L,r11
```

Example 4: using input and output parameters

Multiply two C variables and assign the result to a third C variable. Word registers are necessary for the input and output parameters (constraint `w`, `%0` for `out1`, `%1` for `out2`, `%2` for `in1` and `%3` 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 variables.

```
int in1, in2;
long int out;
void multiply32( void )
{
    unsigned int out1, out2;

    __asm( "CoMUL %2, %3\n"
           "\tCoSTORE %0, MAL\n"
           "\tCoSTORE %1, MAH\n"
           : "=w" (out1), "=w" (out2)
           : "w" (in1), "w" (in2) );

    out =  out1 | (signed long)out2<<16;
}
```

Generated assembly code:

```
; Code generated by C compiler
    movw r11, _in1
    movw r12, _in2
; __asm statement expansion
    CoMUL r11, r12
    CoSTORE r12, MAL
    CoSTORE r11, MAH
; Code generated by C compiler
    movw _out, r12
    movw _out+2, r11
```

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 registers R11 and R12 are reserved registers. You can do this by adding a reserved register list (: "R11","R12"). As you can see in the generated assembly code, registers R11 and R12 are not used (the first register used is R13).

```
int in1, in2;
long int out;
void multiply32( void )
{
    unsigned int out1, out2;

    __asm( "CoMUL %2, %3\n"
           "CoSTORE %0, MAL\n"
           "CoSTORE %1, MAH\n"
           : "=w" (out1), "=w" (out2)
           : "w" (in1), "w" (in2)
           : "R11","R12" );

    out =  out1 | (signed long)out2<<16;
}
```

Generated assembly code:

```
; Code generated by C compiler
movw r13, _in1
movw r14, _in2
; __asm statement expansion
CoMUL r13, r14
CoSTORE r14, MAL
CoSTORE r13, MAH
; Code generated by C compiler
movw _out, r14
movw _out+2, 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(
    "shl %1,#2\n\t"
    "add %2,%1\n\t"
    "mov [%2],%5\n\t"
    "mov %0,%5\n"
    : "=&w" (retvalue), "=w" (par1), "=w" (par2)
    : "1" (par1), "2" (par2), "w" (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:

```
  movw r11,#0x3e8
  movw r12,r11
  movw r13,#_parm

  shl r11,#2
  add r12,r11
  mov [r12],r13
  mov r14,r13

  movw _result,r14
```

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(
    "shl %1,#2\n\t"
    "add %2,%1\n\t"
    "mov [%2],%3\n\t"
    "mov %0,%3\n"
    : "=&w" (retvalue)
    : "w" (par1), "w" (par2), "w" (par3)
);
```

Generated assembly code:

```
  movw r11,#0x3e8
  movw r12,#_parm

  shl r11,#2
  add r11,r11       ; same register, but is expected read-only
  mov [r11],r12
  mov r13,r12
```

```
   movw _result,r13 ; contains unexpected result
```

1.5. 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. For example, you may use __noreturn__ instead of __attribute__((noreturn)).

alias("*symbol*")

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

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

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

const

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

The following kinds of functions should not be declared __const__:

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

• A function that calls a non-const function.

export

You can use __attribute__((export)) to specify that a variable/function has external linkage and should not be removed by the compiler. 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.

malloc

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

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

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

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

noinline

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

always_inline

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

noreturn

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

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

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

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

protect

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

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

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

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

pure

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

section("*section_name*")

You can use `__attribute__((section("name")))` to specify that a function 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 2010 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`.

1.6. 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 pragma-spec pragma-arguments [on | off | default | restore]
```

or:

```
_Pragma( "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

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

alias *symbol=defined_symbol*

Define *symbol* as an alias for *defined_symbol*. It corresponds to an alias directive (.ALIAS) 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 / 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**.

clear_bit / noclear_bit

Same as clear/noclear, except that it only applies to __bit variables.

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

compactmaxmatch {*value* | default | restore}

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

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

constant_memory {*space* | default | restore}

Controls the allocation of constants, automatic initializers and switch tables. The memory space must be one of: __near, __far, __shuge, __huge or model.

See C compiler option **--constant-memory**.

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.

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.

indirect_access {*address*[-*address*],... | default | restore}

Specify address ranges that should only be accessed using an indirect addressing mode.

inline / noinline / smartinline

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

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

#pragma inline
int add( int a, int b )
{
    int i=4;
    return( a + b );
}
#pragma noinline

void main( void )
{
    w = add( 1, 2 );
    z = add( x, y );
}
```

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

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

See also Section 1.12.3, *Inlining Functions: inline*

inline_max_incr / inline_max_size {*value* | default | restore}

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

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

integer_enumeration [on | off | default | restore]

With this pragma the compiler always treats `enum` types as integers. Same as C compiler option **--integer-enumeration**

linear_switch / jump_switch / binary_switch / smart_switch

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

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

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

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

smart_switch   let the compiler decide the switch method used

See also Section 1.11, *Switch Statement*.

mac / nomac

Enable/disable automatic MAC code generation for a function. The pragma works the same as C compiler option **--mac**

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 4.8.2, *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 4.6, *Compiler Optimizations*.

profile [*flags* | default | restore] / endprofile

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

profiling [on | off | default | restore]

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

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

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

ramdata / noramdata

With pragma `ramdata` non-automatic constant data is allocated in both ROM and RAM. At startup RAM is initialized from ROM. This pragma affects `const` variables, string literals, initializers and constants that are allocated in memory. With pragma `noramdata` non-automatic constant data is allocated in ROM only.

See also Section 1.10, *Constant Data*.

romdata / noromdata

With pragma `romdata` the compiler allocates all non-automatic non-constant variables in ROM only. With pragma `noromdata` the variables are allocated in RAM and initialized from ROM at startup.

See also Section 1.8.1, *Initialized Variables*.

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.

savemac / nosavemac

Enable/disable save/restore of MAC-accumulator in a function's prologue/epilogue.

section [*type*=*name* | default | restore] / endsection

Generate code/data in a new section. See Section 1.15, *Section Naming* for more information.

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**.

stack_address_conversion *mode* [default | restore]

Controls how stack addresses are converted. The *mode* can be one of: static, fixed-dpp or dynamic.

See C compiler option **--stack-address-conversion**.

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. The memory space must be one of: __near, __far, __shuge, __huge or model.

See C compiler option **--string-literal-memory**.

tradeoff {*level* | default | restore}

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

warning [*number*[-*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.

1.7. 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 0. The processor accesses data in little-endian.

Macro	Description
__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.
__C166__	Identifies the compiler. You can use this symbol to flag parts of the source which must be recognized by the **c166** compiler only. It expands to 1.
__CORE__	Expands to the name of the core depending on the C compiler options **--cpu** and **--core**. The symbol expands to c16x when no **--cpu** and no **--core** is supplied.
__CORE_*core*__	A symbol is defined depending on the options **--cpu** and **--core**. The *core* is converted to upper case. For example, if **--cpu=xc167ci** is specified, the symbol __CORE_XC16X__ is defined. When no **--core** or **--cpu** is supplied, the compiler defines __CORE_C16X__.
__CPU__	Expands to the name of the CPU supplied with the option **--cpu**. When no **--cpu** is supplied, this symbol is not defined. For example, if **--cpu=xc167ci** is specified, the symbol __CPU__ expands to xc167ci.
__CPU_*cpu*__	A symbol is defined depending on the option **--cpu=***cpu*. The *cpu* is converted to uppercase. For example, if **--cpu=xc167ci** is specified, the symbol __CPU_XC167CI__ is defined. When no **--cpu** is supplied, this symbol is not defined.
__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.
__LINE__	Expands to the line number of the line where this macro is called.
__LITTLE_ENDIAN__	Expands to 1. The processor accesses data in little-endian.
__MODEL__	Identifies the memory model for which the current module is compiled. It expands to a single character constant: 'n' (near), 'f' (far), 's' (shuge) or 'h' (huge).
__NEAR_FUNCTIONS__	Expands to 1 if you used option **--near-functions**, otherwise unrecognized as macro, meaning that huge functions are default.
__PROF_ENABLE__	Expands to 1 if profiling is enabled, otherwise expands to 0.
__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
__SFRFILE__(*cpu*)	This macro expands to the filename of the used SFR file, including the pathname and the < >. The *cpu* is the argument of the macro. For example, if **--cpu=xc167ci** is specified, the macro __SFRFILE__(__CPU__) expands to __SFRFILE__(xc167ci), which expands to <sfr/regxc167ci.sfr>.

Macro	Description
__SILICON_BUG_*num*__	This symbol is defined if the number *num* is defined with the option **--silicon-bug**.
__SINGLE_FP__	Expands to 1 if you used option **--no-double** (Treat 'double' as 'float'), otherwise unrecognized as macro.
__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.
__TASKING_SFR__	Expands to 1 if TASKING `.sfr` files are used. Not defined if you used option **--no-tasking-sfr**.
__TIME__	Expands to the compilation time: "hh:mm:ss"
__USER_STACK__	Expands to 1 if you used option **--user-stack**, otherwise unrecognized as macro.
__USMLIB__	Expands to `__usm` if you used option **--user-stack**, otherwise it expands to `__nousm`. You can use this macro to qualify functions explicitly.
__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).
__VX__	Identifies the VX-toolset C compiler. Expands to 1.

Example

```
#if __MODEL__ == 'f'
/* this part is only for the far memory model */
...
#endif
```

1.8. Variables

1.8.1. Initialized Variables

Automatic initialized variables are initialized (run-time) each time a C function is entered. Normally, this is done by generating code which assigns the value to the automatic variable.

The ISO C standard allows run-time initialization of automatic integral and aggregate types. To support this feature, the C166 C compiler generates code to copy the initialization constants from ROM to RAM each time the function is entered.

There is a lot of existing C source which use static initializations. Static initialized variables normally use the same amount of space in both ROM and RAM. This is because the initializers are stored in ROM and

copied to RAM at start-up. The only exception is an initialized variable residing in ROM, by means of either the `#pragma romdata` or the `const` type qualifier.

```
const char b = 'b';     /* 1 byte  in ROM */
#pragma noromdata       /* default, may be omitted, unless pragma
                           romdata was used before */
int    i = 100;         /* 2 bytes in ROM, 2 bytes in IRAM */
char   a = 'a';         /* 1 byte  in ROM, 1 byte  in IRAM */
char * p = "ABCD";      /* 5 bytes in ROM (for "ABCD") */
                        /* 2 bytes in ROM, 2 bytes in IRAM
                           (for p)*/

#pragma romdata         /* Needed for ROM only allocation */
int    j = 100;         /* 2 bytes in ROM */
char * q = "WXYZ";      /* 5 bytes in ROM (for "WXYZ") */
                        /* 2 bytes in ROM (for p) */
```

1.8.2. Non-Initialized Variables

In some cases there is a need to keep variables unchanged even if power is turned off (see for an example Section 8.7.8, *The Section Layout Definition: Locating Sections*). In these systems some of the RAM is implemented in EEPROM or in a battery-powered memory device. In a simulator environment, clearing non-initialized variables might not be wanted too. To avoid the 'clearing' of non-initialized variables at startup, one of the following things should be performed:

- Define (allocate) these variables in a special C module and compile this module with option **--no-clear**. From Eclipse: From the **Project** menu, select **Properties for**, expand **C/C++ Build**, select **Settings** and open the **Tool Settings** tab, select **C/C++ Compiler » Allocation** and disable the option **Clear non-initialized global variables**.

- Define (allocate) these variables between `#pragma noclear` and `#pragma clear`.

- Use inline assembly to allocate the special variables in a special data section (NOT used by other C variables).

- Make a separate assembly module, containing the allocation of these variables in a special data section.

1.9. Strings

In this context the word 'strings' means the separate occurrence of a string in a C program. So, array variables initialized with strings are just initialized character arrays, which can be allocated in any memory type, and are not considered as 'strings'.

Strings have static storage. The ISO C standard permits string literals to be put in ROM. Because there is no difference in accessing ROM or RAM, the C166 C compiler allocates strings in ROM only. This approach also saves RAM, which can be very scarce in an embedded (single chip) application.

As mentioned before, the C compiler offers the possibility to allocate a static initialized variable in ROM only, when declared with the `const` qualifier or after a `#pragma romdata`. This enables the initialization of a (const) *character array* in ROM:

```
const char romhelp[] = "help";
/* allocation of 5 bytes in ROM only */
```

Or a pointer array in ROM only, initialized with the addresses of strings, also in ROM only:

```
char * const messages[] = {"hello","alarm","exit"};
```

Allocation of string literals

By default the C compiler allocates string literals in the memory model's default memory space. You can overrule this with `#pragma string_literal_memory`:

```
#pragma string_literal_memory space
```

The *space* must be one of: `__near`, `__far`, `__shuge`, `__huge` or `model`. Instead of this pragma you can also use the equivalent command line option **--string_literal_memory**.

String literals as use in:

```
char * s = "string";
```

or:

```
printf("formatter %s\n", "string");
```

are affected by this pragma/option.

Example:

```
#pragma string_literal_memory __huge   /* allocate strings in __huge memory */
__huge char * txt = "text1";
```

1.10. Constant Data

By default `const` variables, string literals, initializers, switch tables and constants allocated in memory are stored in ROM only. You can change this behavior with `#pragma ramdata`. When this pragma is active, `const` variables, string literals, initializers and constants allocated in memory will be allocated in ROM and RAM. At startup RAM will be initialized from ROM, in the same way as initialized variables. To achieve this the `"init"` instead of the `"romdata"` section attribute is used. The ROM copy will always be located in the `__shuge` memory space.

Example:

```
#pragma ramdata
const char * const p = "copied to RAM";
      /* both p and the string literal are copied from ROM to RAM */

near__1_str  .section  near, byte, init, new
_$1$str      .label    nearbyte
      .db    0x63
      .db    0x6f
      .db    0x70
```

```
        .db      0x69
        .db      0x65
        .db      0x64
        .db      0x20
        .db      0x74
        .db      0x6f
        .db      0x20
        .db      0x52
        .db      0x41
        .db      0x4d
        .db      0x0
        ; End of section

near_p        .section      near, init, new
        .global  _p
_p      .label   nearword
        .dw      (_$1$str & 0xffff)
```

1.11. Switch Statement

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

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

`#pragma smart_switch` is the default of the compiler. The compiler tries to use the switch method which uses the least space in ROM (table size in ROMDATA plus code to do the indexing). With the C compiler option **--tradeoff** you can tell the compiler to emphasis more on speed than on ROM size.

Especially for large switch statements, the jump table approach executes faster than the binary search table approach. Also the jump table has a predictable behavior in execution speed: independent of the switch argument, every case is reached in the same execution time.

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

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

`#pragma linear_switch`	force jump chain code
`#pragma jump_switch`	force jump table code
`#pragma binary_switch`	force binary search table code
`#pragma smart_switch`	let the compiler decide the switch method used

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 jump_switch

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

1.12. Functions

Near and huge functions

By default functions are huge. With the C compiler option **--near-functions** you can set the default to use near functions. But you can also use the __near or __huge function pointer qualifiers.

__near   Define function called with intra-segment calls. The sections generated for __near functions are grouped in a group called __near_functions.

__huge   Define function called with inter-segment calls.

Example:

```
__near nfunc(void){ /* a near function */ }
```

Function pointers

The size of a pointer to a __near function is two bytes. The size of a pointer to a __huge function is four bytes. Both pointers are aligned on two bytes.

Note that you cannot cast a near function pointer to a huge function pointer due to possible run-time errors.

Function call return addresses

The compiler uses a 'user stack' to pass parameters and to allocate variables and temporary storage. The function return addresses are placed on the system stack by the processor with a call instruction. With the C compiler option **--user-stack** function return addresses are placed on the user stack. The code compaction optimization (**-Or**) has no effect for functions with the return address on the user stack.

Instead of the option **--user-stack**, you can use the __usm or __nousm function pointer qualifiers.

__usm   Use the user stack for function call return addresses.

__nousm   Use the system stack for function call return addresses.

1.12.1. Calling Convention

Parameter passing

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

The following conventions are used when passing parameters to functions.

Registers available for parameter passing are USR0, R2, R3, R4 R5, R11, R12, R13 and R14. Parameters <= 64 bit are passed in registers except for 64-bit structures:

Parameter Type	Registers used for parameters
1 bit	USR0, R2.0..15, R3.0..15, R4.0..15, R5.0..15
8 bit	RL2, RH2, RL3, RH3, RL4, RH4, RL5, RH5
16 bit	R2, R3, R4, R5, R11, R12, R13, R14
32 bit	R2R3, R4R5, R11R12, R13R14
64 bit	R2R3R4R5, R11R12R13R14

The parameters are processed from left to right. The first not used and fitting register is used. Registers are searched for in the order listed above. When a parameter is > 64 bit, or all registers are used, parameter passing continues on the stack. The stack grows from higher towards lower address, each parameter on the stack is stored in little-endian. The first parameter is pushed at the lowest stack address. The alignment on the stack depends on the data type as listed in Section 1.1, *Data Types*

Example with three arguments:

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

a (first parameter) is passed in registers R2.

b (second parameter) is passed in registers R4R5.

c (third parameter) is passed in registers R3.

Variable argument lists

Functions with a variable argument list must push all parameters after the last fixed parameter on the stack. The normal parameter passing rules apply for all fixed parameters.

Function return values

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

USR0, R2, R3, R4 and R5 are used for return values <=64 bit:

Return Type	Register
1 bit	USR0

Return Type	Register
8 bit	RL2
16 bit	R2
32 bit	R2R3
64 bit	R2R3R4R5

The return registers have an overlap with the parameter registers, which yields more efficient code when passing arguments to child functions.

Return values > 64 bits are returned in a buffer, allocated on the stack. The caller must pass a pointer to the return buffer in the last parameter register (R14). It is the caller's responsibility to allocate and release the space used for the return buffer. The callee will put the return value in the allocated buffer.

Stack usage

The stack on the C166 consists of a system stack and a user stack. The system stack is used for the return addresses and for data explicitly pushed with the PUSH instruction. The compiler usually does not push anything on the system stack, with exception to interrupt functions . The user stack is used for parameter passing, allocation of automatics and temporary storage. The compiler uses R15 as user stack pointer. The data on the stack is aligned depending on the data type as listed in Section 1.1, *Data Types*. The stack pointer itself is always aligned at 16-bit. In the Super10/XC16x a user stack is allocated for each local bank. The user stack grows from high to low. The user stack is always located in near memory, the maximum size depends on the chosen memory model. The DPP register used for the user stack is determined at link time.

The stack pointer always refers to the last occupied slot. Meaning that the stack pointer first has to be decreased before data can be stored. A typical stack frame is outlined in the following picture:
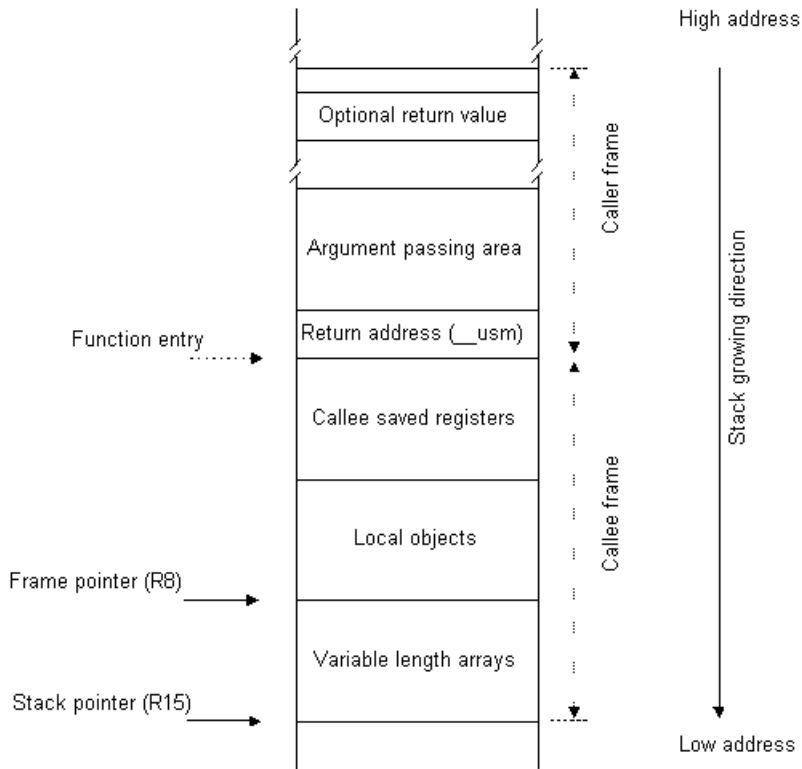
Before a function call, the caller pushes the required parameters on the stack. This area is called the argument passing area. For user stack functions the return address is saved on the user stack. After the call has been made, the callee will save the used callee-saved registers in the "callee saved" area. Next, the space for the local objects is allocated. After this, variable length arrays (VLAs) can be allocated. If VLAs are used in a function, register R8 is used to access the local objects and stack parameters. If no VLAs are used, R8 is available for other purposes. When the called function returns an object > 64 bit on the stack, the caller must reserve a stack area to hold the return value. After the function call, the caller must release this stack area. This also applies to the argument passing area. After the stack frame has been built, the stack pointer points to the argument passing area.

1.12.2. Register Usage

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

Register	Class	Purpose
USR0	caller saves	Parameter passing and return values
R0, RL0, RH0	callee saves	Automatic variables
R1, RL1, RH1	callee saves	Automatic variables
R2, RL2, RH2	caller saves	Parameter passing and return values

Register	Class	Purpose
R3, RL3, RH3	caller saves	Parameter passing and return values
R4, RL4, RH4	caller saves	Parameter passing and return values
R5, RL5, RH5	caller saves	Parameter passing and return values
R6, RL6, RH6	callee saves	Automatic variables
R7, RL7, RH7	callee saves	Automatic variables
R8	callee saves	Automatic variables, User stack frame pointer
R9	callee saves	Automatic variables
R10	callee saves	Automatic variables
R11	caller saves	Parameter passing
R12	caller saves	Parameter passing
R13	caller saves	Parameter passing
R14	caller saves	Parameter passing, return buffer pointer
R15	dedicated	User stack pointer

The registers are classified: caller saves, callee saves and dedicated.

caller saves   These registers are allowed to be changed by a function without saving the contents. Therefore, the calling function must save these registers when necessary prior to a function call.

callee saves   These registers must be saved by the called function, i.e. the caller expects them not to be changed after the function call.

dedicated   The user stack pointer register R15 is dedicated.

The user stack frame pointer register R8 is used for functions containing variable length arrays.

Registers R0, R1, R2 and R3 can be used directly in an arithmetic instruction like: ADD Rx, [R0].

1.12.3. 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**.

1.12.4. Interrupt Functions

The TASKING C compiler supports a number of function qualifiers and keywords to program interrupt service routines (ISR). An *interrupt service routine* (or: interrupt function, interrupt handler, exception handler) is called when an interrupt event (or: *service request*) occurs.

Defining an interrupt service routine: __isr, __interrupt()

You can use the type qualifier __isr to declare a function as an interrupt service routine, but this does not bind the function to an interrupt vector. You can assign an unbound function to an interrupt vector in the linker LSL file. With the function type qualifier __interrupt() you can bind the function to a specific vector in the C source. The function type qualifier __interrupt() takes one interrupt number (0..127) as an argument. The linker generates the sections with the vectors of the specified interrupt numbers.

For backward compatibility, __interrupt(-1) is the same as __isr.

The __interrupt() function qualifier implies the __isr type qualifier.

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

```
void __isr __interrupt(interrupt_number)
isr( void )
{
...
}
```

The __isr type qualifier must also be used when a pointer to an interrupt function is declared.

For example:

```
void __interrupt( 7 ) serial_receive( void )
{   /* __isr is added automatically by __interrupt() */
  ...
}

void __isr  unbound_isr( void ) /* same as: __interrupt( -1 )          */
{                               /* unbound ISR can be bound in LSL file */
  ...
}

extern void __isr  external_isr( void );
  /* reference to external interrupt function, vector number irrelevant */
void __isr (*pisr)( void ) = external_isr;
  /* declare pointer to interrupt function */
```

GPRs are pushed on the system stack, unless you use the __registerbank() qualifier.

Interrupt frame: __frame()

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

```
void __interrupt(interrupt_number)
     __frame(reg[, reg]...) isr( void )
{
```

```
...
}
```

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

Example:

```
void __interrupt(8) __frame(MDL, MDH ) foo (void)
{
 ...
}
```

You can also use the `__frame()` qualifier in conjunction with the `__registerbank()` qualifier to add code for the context switch in the interrupt frame.

When you do not want the interrupt frame (saving/restoring registers) to be generated you can use the C compiler option **--no-frame**. In that case you will have to specify your own interrupt frame. For this you can use the inline capabilities of the compiler.

Register bank switching: __registerbank()

It is possible to assign a new register bank to an interrupt function, which can be used on the processor to minimize the interrupt latency because registers do not need to be pushed on stack. You can switch register banks with the `__registerbank()` function qualifier. The syntax is:

```
void __interrupt(interrupt_number)
  __registerbank(["regbank" | localbank[,"regbank"]])
isr( void )
{
...
}
```

  regbank   The string specifies the name of a global register bank to be used. The compiler generates a section for the register bank. The compiler assumes that the BANK field in the PSW register already selects a global register bank.

localbank   The number of the local register bank to be used. With a negative number, the compiler assumes that the register bank switch is done automatically by the processor. With a positive number, the compiler generates code to select the local register bank. With zero, the compiler generates code to select a global register bank. In the last case, an extra argument can be used to specify the name of the global register bank. If omitted, the compiler will generate a name. The following numbers are available:

-3   Use local register bank 3 but assume the hardware automatically switches the register bank upon interrupt.

-2   Use local register bank 2 but assume the hardware automatically switches the register bank upon interrupt.

-1   Use local register bank 1 but assume the hardware automatically switches the register bank upon interrupt.

0   Use global register bank as usual.

1   Use local register bank 1 and emit instruction in interrupt frame to select the correct local register bank.

2   Use local register bank 2 and emit instruction in interrupt frame to select the correct local register bank.

3   Use local register bank 3 and emit instruction in interrupt frame to select the correct local register bank.

For the Super10XC16x a user stack is allocated for each bank. The user stack pointers are initialized in the C startup code. For the user stack in the global register bank you can use the linker label `_lc_ub_user_stack`. For the local register banks 1, 2 and 3 use linker labels `_lc_ub_user_stack1`, `_lc_ub_user_stack2` and `_lc_ub_user_stack3` respectively.

When no *regbank*-argument is supplied the compiler generates and uses a register bank with the name _$*fname*_regbank, where *fname* represents the name of the interrupt function.

When the `__registerbank()` qualifier is omitted, the compiler will save the GPRs on the system stack.

When the `__registerbank()` qualifier, that selects a global register bank, is used on the reset vector (`__interrupt( 0 )`), the context pointer will be initialized, instead of being saved.

1.12.5. Intrinsic Functions

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

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.

Many CoXXX instructions are automatically generated if a special sequence is recognized. For example,

```
__CoLOAD( arg1 );
__CoABS();
```

generates the CoABS op1, op2 instruction.

```
__CoMUL( arg1, arg2 );
__CoRND();
```

generates the CoMUL op1, op2, rnd instruction.

```
__CoSUB( arg1 );
__CoNEG();
```

generates the CoSUBR op1, op2 instruction.

__CoABS

```
void __CoABS( void );
```

Use the CoABS instruction to change the MAC accumulator's contents to its absolute value.

__CoADD

```
void __CoADD( long x );
```

Use the CoADD instruction to add a 32-bit value to the MAC accumulator.

__CoADD2

```
void __CoADD2( long x );
```

Use the CoADD2 instruction to add a 32-bit value, multiplied by two, to the MAC accumulator.

__CoASHR

```
void __CoASHR( unsigned int count );
```

Use the CoASHR instruction to (arithmetic) shift right the contents of the MAC accumulator count times.

The CoASHR instruction has a maximum value for count. Check your CPU manual for the CoASHR behavior for large arguments.

__CoCMP

```
unsigned int __CoCMP( long x );
```

Inline code is generated by the C compiler to compare the MAC accumulator contents with a 32-bit value. The returned value is a copy of the MSW register.

__CoLOAD

```
void __CoLOAD( long x );
```

Use the CoLOAD instruction to copy a 32-bit value to the MAC accumulator.

__CoLOAD2

```
void __CoLOAD2( long x );
```

Use the CoLOAD2 instruction to copy a 32-bit value, multiplied by two, to the MAC accumulator.

__CoMAC

```
void __CoMAC( int x, int y );
```

Use the CoMAC instruction to add the multiplication result of two signed 16-bit values to the MAC accumulator.

__CoMACsu

```
void __CoMACsu( int x, unsigned int y );
```

Use the CoMACsu instruction to add the multiplication result of a signed 16-bit value with an unsigned 16-bit value to the MAC accumulator.

__CoMACu

```
void __CoMACu( unsigned int x, unsigned int y );
```

Use the CoMACu instruction to add the multiplication result of two unsigned 16-bit values to the MAC accumulator.

__CoMACus

```
void __CoMACu( unsigned int x, signed int y );
```

Use the CoMACus instruction to add the multiplication result of an unsigned 16-bit value with a signed 16-bit value to the MAC accumulator.

__CoMAC_min

```
void __CoMAC_min( int x, int y );
```

Use the CoMAC- instruction to subtract the multiplication result of two signed 16-bit values from the MAC accumulator.

__CoMACsu_min

```
void __CoMACsu_min( int x, unsigned int y );
```

Use the CoMACsu- instruction to subtract the multiplication result of a signed 16-bit value with an unsigned 16-bit value from the MAC accumulator.

__CoMACu_min

```
void __CoMACu_min( unsigned int x, unsigned int y );
```

Use the CoMACu- instruction to subtract the multiplication result of two unsigned 16-bit values from the MAC accumulator.

__CoMACus_min

```
void __CoMACus_min( unsigned int x, signed int y );
```

Use the CoMACus- instruction to subtract the multiplication result of an unsigned 16-bit value with a signed 16-bit value from the MAC accumulator.

__CoMAX

```
void __CoMAX( long x );
```

Use the CoMAX instruction to change the MAC accumulator's contents if its value is lower than the argument's value.

__CoMIN

```
void __CoMIN( long x );
```

Use the CoMIN instruction to change the MAC accumulator's contents if its value is higher than the argument's value.

__CoMUL

```
void __CoMUL( int x, int y );
```

Use the CoMUL instruction to store the multiplication result of two signed 16-bit values in the MAC accumulator.

__CoMULsu

```
void __CoMULsu( int x, unsigned int y );
```

Use the CoMULsu instruction to store the multiplication result of a signed 16-bit value with an unsigned 16-bit value in the MAC accumulator.

__CoMULu

```
void __CoMULu( unsigned int x, unsigned int y );
```

Use the CoMULu instruction to store the multiplication result of two unsigned 16-bit values in the MAC accumulator.

__CoMULus

```
void __CoMULus( unsigned int x, signed int y );
```

Use the CoMULus instruction to store the multiplication result of an unsigned 16-bit value with a signed 16-bit value in the MAC accumulator.

__CoMUL_min

```
void __CoMUL_min( int x, int y );
```

Use the CoMUL- instruction to store the negated multiplication result of two signed 16-bit values in the MAC accumulator.

__CoMULsu_min

```
void __CoMULsu_min( int x, unsigned int y );
```

Use the CoMULsu- instruction to store the negated multiplication result of a signed 16-bit value with an unsigned 16-bit value in the MAC accumulator.

__CoMULu_min

```
void __CoMULu_min( unsigned int x, unsigned int y );
```

Use the CoMULu- instruction to store the negated multiplication result of two unsigned 16-bit values in the MAC accumulator.

__CoMULus_min

```
void __CoMULus_min( unsigned int x, signed int y );
```

Use the CoMULus- instruction to store the negated multiplication result of an unsigned 16-bit value with a signed 16-bit value in the MAC accumulator.

__CoNEG

```
void __CoNEG( void );
```

Use the CoNEG instruction to change the MAC accumulator's contents to its negated value.

__CoNOP

```
void __CoNOP( void );
```

A CoNOP instruction is generated.

__CoRND

```
void __CoRND( void );
```

Use the CoRND semi-instruction to change the MAC accumulator's contents to its rounded value.

__CoSHL

```
void __CoSHL( unsigned int count );
```

Use the CoSHL instruction to shift left the contents of the MAC accumulator `count` times.

The CoSHL instruction has a maximum value for `count`. Check your CPU manual for the CoSHL behavior for large arguments.

__CoSHR

```
void __CoSHR( unsigned int count );
```

Use the CoSHR instruction to (logical) shift right the contents of the MAC accumulator `count` times.

The CoSHR instruction has a maximum value for `count`. Check your CPU manual for the CoSHR behavior for large arguments.

__CoSTORE

```
long __CoSTORE( void );
```

Use the CoSTORE instruction to retrieve the 32-bit value, stored in the MAC accumulator MAH and MAL.

__CoSTOREMAH

```
int __CoSTOREMAH( void );
```

Use the CoSTORE instruction to retrieve the 16-bit value, stored in MAH.

__CoSTOREMAL

```
int __CoSTOREMAL( void );
```

Use the CoSTORE instruction to retrieve the 16-bit value, stored in MAL.

__CoSTOREMAS

```
int __CoSTOREMAS( void );
```

Use the CoSTORE instruction to retrieve the 16-bit value, stored in MAS.

__CoSTOREMSW

```
int __CoSTOREMSW( void );
```

Use the CoSTORE instruction to retrieve the 16-bit value, stored in MSW.

__CoSUB

```
void __CoSUB( long x );
```

Use the CoSUB instruction to subtract a 32-bit value from the MAC accumulator.

__CoSUB2

```
void __CoSUB2( long x );
```

Use the CoSUB2 instruction to subtract a 32-bit value, multiplied by two, from the MAC accumulator.

__alloc

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

Allocate memory on the user stack. Returns a pointer to space in external memory of size bytes length. NULL if there is not enough space left.

__dotdotdot__

```
char * __dotdotdot__( void );
```

Variable argument '...' operator. Used in library function `va_start()`. Returns the stack offset to the variable argument list.

__free

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

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

__getsp

```
__near void * volatile __getsp( void );
```

Get the value of the user stack pointer. Returns the value of the user stack pointer.

__setsp

```
void volatile __setsp( __near void * value );
```

Set the value of the user stack pointer to `value`.

__get_return_address

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

Used by the compiler for profiling when you compile with the option **--profile**. Returns the return address of a function.

__rol

```
unsigned int __rol( unsigned int operand,
                    unsigned int count );
```

Use the ROL instruction to rotate `operand` left `count` times.

__ror

```
unsigned int __ror( unsigned int operand,
                    unsigned int count );
```

Use the ROR instruction to rotate `operand` right `count` times.

__testclear

```
__bit __testclear( __bit semaphore );
```

Read and clear semaphore using the JBC instruction. Returns 0 if semaphore was not cleared by the JBC instruction, 1 otherwise.

__testset

```
__bit __testset( __bit semaphore );
```

Read and set semaphore using the JNBS instruction. Returns 0 if semaphore was not set by the JNBS instruction, 1 otherwise.

__bfld

```
void __bfld( volatile unsigned int __unaligned * operand,
             unsigned short mask, unsigned short value );
```

Use the BFLDL/BFLDH instructions to assign the constant `value` to the bit-field indicated by the constant `mask` of the bit-addressable `operand`.

__getbit

```
__bit __getbit( operand, bitoffset );
```

Return the bit at `bitoffset` of the bit-addressable `operand` for usage in bit expressions.

__putbit

```
void __putbit( __bit value, operand, bitoffset );
```

Assign `value` to the bit at `bitoffset` of the bit-addressable `operand`.

__int166

```
void __int166( intno );
```

Execute the C166/ST10 software interrupt specified by the interrupt number `intno` via the software trap (TRAP) instruction. `__int166( 0 );` emits an SRST (Software Reset) instruction. `__int166( 8 );` emits an SBRK (Software Break) instruction (only for super10/super10m345/xc16x cores).

__idle

```
void __idle( void );
```

Use IDLE instruction to enter the idle mode. In this mode the CPU is powered down while the peripherals remain running.

__nop

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

A NOP instruction is generated, before and behind the nop instruction the peephole is flushed.

__prior

```
unsigned int __prior( unsigned int value );
```

Use PRIOR instruction to prioritize `value`.

__pwrdn

```
void __pwrdn( void );
```

Use PWRDN instruction to enter the power down mode. In this mode, all peripherals and the CPU are powered down until an external reset occurs.

__srvwdt

```
void __srvwdt( void );
```

Use SRVWDT instruction to service the watchdog timer.

__diswdt

```
void __diswdt( void );
```

Use DISWDT instruction to disable the watchdog timer.

__enwdt

```
void __enwdt( void );
```

Use ENWDT instruction to enable the watchdog timer.

__einit

```
void __einit( void );
```

Use EINIT instruction to end the initialization.

__mul32

```
long __mul32( int x, int y );
```

Use MUL instruction to perform a 16-bit by 16-bit signed multiplication and returning a signed 32-bit result. The overflow bit V is set by the CPU when the result cannot be represented in an `int` data type.

__mulu32

```
unsigned long __mulu32( unsigned int x,
                        unsigned int y );
```

Use MULU instruction to perform a 16-bit by 16-bit unsigned multiplication and returning a unsigned 32-bit result. The overflow bit V is set by the CPU when the result cannot be represented in an `int` data type.

__div32

```
int __div32( long x, int y );
```

Use DIVL instructions to perform a 32-bit by 16-bit signed division and returning a signed 16-bit result. The overflow bit V is set by the CPU when the result cannot be represented in an `int` data type or when the divisor yy was zero.

__divu32

```
unsigned int __divu32( unsigned long x,
                       unsigned int y );
```

Use DIVLU instructions to perform a 32-bit by 16-bit unsigned division and returning an unsigned 16-bit result. The overflow bit V is set by the CPU when the result cannot be represented in an `int` data type or when the divisor y was zero.

__mod32

```
int __mod32( long x, int y );
```

Use DIVL instructions to perform a 32-bit by 16-bit signed modulo and returning a signed 16-bit result. The overflow bit V is set by the CPU when the quotient cannot be represented in an `int` data type or when the divisor y was zero.

__modu32

```
unsigned int __modu32( unsigned long x,
                       unsigned int y );
```

Use DIVLU instructions to perform a 32-bit by 16-bit unsigned modulo and returning a unsigned 16-bit result. The overflow bit V is set by the CPU when the quotient cannot be represented in an `int` data type or when the divisor y was zero.

__pag

```
unsigned int __pag( void * p );
```

Inline code is generated by the C compiler to get the 10-bit page number of pointer `p`

__pof

```
unsigned int __pof( void * p );
```

Inline code is generated by the C compiler to get the 14-bit page offset of pointer `p`

__seg

```
unsigned int __seg( void * p );
```

Inline code is generated by the C compiler to get the 8-bit segment number of pointer `p`

__sof

```
unsigned int __sof( void * p );
```

Inline code is generated by the C compiler to get the 16-bit segment offset of pointer `p`

__mkfp

```
void __far *  __mkfp( unsigned int pof,
                      unsigned int pag );
```

Inline code is generated by the C compiler to make a far pointer from a page offset `pof` and page number `pag`. The arguments `pag` and `pof` are expected to be in a valid range.

__mkhp

```
void __huge *  __mkhp( unsigned int sof,
                       unsigned int seg );
```

Inline code is generated by the C compiler to make a huge pointer from a segment offset `sof` and segment number `seg`. The arguments `sof` and `seg` are expected to be in a valid range.

__mksp

```
void __shuge * __mksp( unsigned int sof,
                       unsigned int seg );
```

Inline code is generated by the C compiler to make a shuge pointer from a segment offset `sof` and segment number `seg`. The arguments `sof` and `seg` are expected to be in a valid range.

__sat

```
void __sat( void );
```

Enable saturation. The compiler will automatically save/restore the MCW register in a functions prologue/epilogue (both regular functions and ISRs).

__nosat

```
void __nosat( void );
```

Disable saturation. The compiler will automatically save/restore the MCW register in a functions prologue/epilogue (both regular functions and ISRs).

__scale

```
void __scale( void );
```

Enable scaler. The compiler will automatically save/restore the MCW register in a functions prologue/epilogue (both regular functions and ISRs).

__noscale

```
void __noscale( void );
```

Disable scaler. The compiler will automatically save/restore the MCW register in a functions prologue/epilogue (both regular functions and ISRs).

1.13. Floating-Point Trapping

Two sets of floating-point libraries are delivered with the compiler, one with a floating-point trapping mechanism and one without a floating-point trapping mechanism (Chapter 13, *Libraries* explains the naming conventions).

The floating-point libraries with a trapping mechanism call a trapping routine which is in module `trap.obj`. You can replace this routine with your own trapping routine, or link your own trap routine to your application. By default, the trapping routine as delivered with the floating-point libraries will never return. The infinite loop on a public label called `__fptraploop` is easy to find in a debug session. For the source of the trapping routine, see file `trap.src` in the `lib/src` directory of the product.

To see an example of how floating-point traps can be handled import the `fptrap` example project. In the example all the possible exceptions are generated. You can use this example to write a program which handles the exception without "hanging" the program.

A floating-point routine calls the trap routine if an error condition occurs. The type of error is specified by a trap code which is passed via register R1 to the trap routine. The result of a floating-point operation is not undefined in an error situation. On error the result will be a special floating-point number, such as infinite, not a number etc., except when a floating point underflow or overflow occurs.

The following table lists all the trap codes and the corresponding error description and result:

Error Description	Trap code	Result float/integer
Undefined float	1	NAN (float result)
Divide by zero	2	+INF or -INF or NAN (float result)
Integer overflow	3	0x7FFF or 0x8000 (integer result)
Floating overflow	4	+INF or -INF (float result)

Error Description	Trap code	Result float/integer
Floating underflow	8	0.0 (float result)
Conversion error	32	0 (integer result)
INF Infinite which is the largest absolute floating-point number.		
NAN Not A Number, special notation for undefined floating point number.		

1.13.1. Handling Floating-Point Traps in a C Application

This section explains how program execution can be continued after a floating-point trap and how floating-point trap codes are passed from the floating-point trap handler to a C application.

Only the floating-point libraries which perform floating-point trapping contain a floating-point trap stub. This floating-point trap stub loops infinitely, which is very helpful when you want to find a bug in your application. But when it is expected or allowed or even wanted that floating-point operations generate results that are out of range, then program execution must continue after entering the floating-point trap handler.

It is not possible to simply return from the floating-point trap handler, because the floating-point accumulator(s) contain a value which is out of range. In the same floating-point operation or else in a next floating-point operation there will be another call to the floating-point trap handler, because the value in the floating-point accumulator(s) remain out of range. This results in a succession of floating-point traps.

It is impossible to assign a value to the floating-point accumulator(s) which is in range and then continue program execution. If you try to assign a value to the floating-point accumulators the result will always be undefined.

Interpretation of the error condition in the floating-point trap handler and then continuing the floating-point operation will result in most cases in a new error condition or unpredictable result. So, this is not a good solution to handle floating-point error situations.

It is better to stop immediately the floating-point operation which causes the floating-point trap, by returning back to your application and there decide what to do with the floating-point error condition. Therefore, you have to predefine an environment in your application to return to. Simply jumping back is not possible because the system stack and user stack are then corrupted. The floating-point trap code must also be returned to the application to examine the cause of the trap.

An environment to return to in an application can be saved with the C library function `setjmp`. The C library function `longjmp` can be used in the floating-point trap handler to return immediately to this saved environment. The `longjmp` restores the stack pointers, jumps back and passes the trap code to be processed.

The file `fptrap.c` in the `fptrap` example project delivered with the product shows how to save an environment with `setjmp`. The assembly listing of the floating-point trap handler in the file `trap.src` in `lib/src` shows how `longjmp` is used to return to the saved environment.

There are several ways to write a C function which handles floating-point traps using `setjmp` and `longjmp`. Always keep in mind that the `longjmp` function restores the environment saved by the most recent

invocation of the `setjmp` function. And the environment must be saved before the `longjmp` function is called by the floating-point trap handler, else program execution will be undefined.

The floating-point trap handler checks if an environment is set in `__FP_ENV` to return to. When the return address contains a NULL pointer it is supposed that there is no environment set and the trap handler continues looping infinitely. When a return address is set, the address of the jump buffer `__FP_ENV` and the trap code are passed to `longjmp`. Calling the `longjmp` function at the end of the trap handler restores the environment saved in `__FP_ENV`.

The data section containing the floating-point jump buffer `__FP_ENV` is cleared at startup. The initialization codes for it are stored in the `near_libfp` section.

There are two entry points available in the floating-point trap handler, one for double precision floating-point functions causing a trap (`__fptrap8`), and one for single precision floating-point functions causing a trap (`__fptrap4`). This default trap handler is precision independent, but if you want to write a trap handler for each precision you need these two entry points.

You can use your own floating-point trap handler by linking the object module, overruling the floating-point trap handler of the floating-point library. Or you can replace the floating-point trap object module in the floating-point library with the object module of your own floating-point trap handler.

1.13.2. IEEE-754 Compliant Error Handling

When you use the floating-point libraries without trapping, the routines continue calculation with erroneous input values. This behavior is not conforming to the IEEE-754 standard, but does deliver the highest speed because the input value checking is omitted.

If your application requires IEEE-754 compliant handling of erroneous input values, you should use the trapping version of the floating-point libraries. But if you do not want to handle the error conditions with a trap routine, but just continue calculation conform to IEEE-754, you can provide an empty trap function. You can add the following trap handling code to your application to achieve this:

```
#include <setjmp.h>

#pragma noclear
jmp_buf __FP_ENV;

void __fptrap8( void ) /* double precision */
{
}

void __fptrap4( void ) /* single precision */
{
}
```

1.14. MAC Unit Support

The C166 compiler supports the MAC-unit in the XC16x/Super10 core in four ways:

1. Code generation directly from native C

2. Manual qualification.

3. Intrinsic functions.

4. Evaluation of a single expression.

1.14.1. MAC Code Generation from Native C

Implementation

In the XC16x/Super10 cores, the MAC-unit basically consists of three SFRs that build a single accumulator and an instruction set that operates upon it. Because there is only one accumulator the risk of spilling is high. To spill the accumulator to GPRs, three moves are needed, and another three for the restore. This is expensive. Furthermore, in terms of code size the MAC instruction set is not always the cheapest.

To generate code for the MAC unit, the compiler searches for local objects of type (unsigned) long. This type is chosen, because it is closest to the 40-bit accumulator.

Next, the compiler analyzes the code associated with the objects found. For each operation, the compiler estimates the costs for using the MAC instruction set, compared to the non-MAC instruction set.

The MAC instruction set uses the accumulator as an input, as well as to store the result. Therefore compound expressions with an operation that maps well upon the MAC instruction set will turn out to be the most beneficial. Other operations may be used, but they will have a negative effect upon the overall costs of the object.

As described above there is only one accumulator and spilling is expensive. Therefore, the next step is to perform lifetime analysis upon the selected objects. Using the lifetime analysis and the computed costs, object(s) that are the most beneficial and that do not have overlapping lifetimes are preferred for allocation in the accumulator. This avoids spilling.

Automatic allocation is done by assigning the __mac qualifier to an object. Example

```
long mac( const short * a, const short * b, long sqr, long * sum )
{
    int  i;                 // loop counter
    long dotp = *sum;       // accumulator

    for (i = 0; i < 150; i++)
    {
        dotp += (long)b[i] * a[i];
        sqr  += (long)b[i] * b[i];
    }
    *sum = dotp;
    return sqr;
}
```

When compiled with**--core=xc16x --mac --no-savemac**, this results in:

```
_mac    .proc   far
    ; mac.c       11  {
    ; mac.c       12      int  i;              // loop counter
```

```
    ; mac.c        13     long dotp = *sum;  // accumulator
        movw    r12,r11
    ; mac.c        14
    ; mac.c        15     for (i = 0; i < 150; i++)
        movw    MRW,#0x95
        movw    r13,[r12+]
        CoLOAD  r13,[r12]
_2:
    ; mac.c        16     {
    ; mac.c        17         dotp += (long)b[i] * a[i];
        movw    r12,[r3+]
-usr1   CoMAC   r12,[r2+]
    ; mac.c        18         sqr  += (long)b[i] * b[i];
        mul     r12,r12
        addw    r4,MDL
        addcw   r5,MDH
        jmp     cc_nusr1,_2
    ; mac.c        19     }
    ; mac.c        20
    ; mac.c        21     *sum = dotp;
        CoSTORE [r11+],MAL
        CoSTORE [r11],MAH
    ; mac.c        22     return sqr;
        movw     r2,r4
        movw     r3,r5
    ; mac.c        23  }
        ret
```

As you can see, the compiler has chosen to allocate variable 'dotp' in the accumulator.

Operation costs

The costs of operations can be measured in two ways:

1. Code size.

2. Number of cycles.

When optimize for size (**-t4**) is selected, costs will be computed using the code size only. When optimize for speed (**-t0**) is selected only the number of cycles will be taken into account. In the latter case, the compiler will multiply the costs by the estimated number of loop iterations.

The size in bytes and cycles are weighed with the trade-off setting:

Trade-off value	Time	Size
-t0	100%	0%
-t1	75%	25%
-t2	50%	50%
-t3	25%	75%

Trade-off value	Time	Size
-t4	0%	100%

The estimated execution frequency of an instruction is multiplied by the number of cycles.

The MAC unit can perform complex operations in one cycle, which is in most cases faster than using the non-MAC instruction set. Therefore, the MAC instruction set is more useful when speed optimization is selected. For example, the code for shifting a long to the left is:

```
          size   cycles                          size   cycles
          ----   ------                          ----   ------
  CoSHL #1    4       1        addw  r2,r2           2        1
                               addcw r3,r3           2        1
          ==== + ====== +                       ==== + ====== +
  total       4       1                            4        2
```

As you can see, it will not be beneficial to use the MAC instruction here when optimizing for code size, however when optimizing for speed it is possible to save a cycle by using the MAC instruction set.

1.14.2. Manual MAC Qualification: __mac

With the keyword __mac you can allocate an automatic object in the MAC accumulator. The __mac keyword is advisory to the compiler. It is only honoured for plain automatics and parameter objects of type (unsigned) long. The object cannot be volatile, and it is not allowed to take the address of the object. The compiler will also never assign the __mac qualifier automatically if these restrictions are not met. The compiler will never automatically choose an object for MAC allocation if it has an overlapping lifetime with manually qualified objects. When the __mac keyword is ignored, the compiler generates a warning.

Example:

```
long mac( const short * a, const short * b, __mac long sqr, long * sum )
{
    int  i;                 // loop counter
    long dotp = *sum;       // accumulator

    for (i = 0; i < 150; i++)
    {
        dotp += (long)b[i] * a[i];
        sqr  += (long)b[i] * b[i];
    }
    *sum = dotp;
    return sqr;
}
```

This is the same example as the previous example, with the exception that object 'sqr' has now been qualified explicitly. When compiled with**--core=xc16x --mac --no-savemac**, this results in:

```
_mac    .proc   far
    ; mac.c       11  {
    ; mac.c       12     int  i;              // loop counter
    ; mac.c       13     long dotp = *sum;    // accumulator
```

```
        movw    r12,r11
        CoLOAD  r4,r5
    ; mac.c       14
    ; mac.c       15      for (i = 0; i < 150; i++)
        movw    MRW,#0x95
        movw    r13,[r12+]
        movw    r12,[r12]
_2:
    ; mac.c       16        {
    ; mac.c       17            dotp += (long)b[i] * a[i];
        movw    r14,[r3]
        movw    r4,[r2+]
        mul     r14,r4
-usr1   CoMAC   r14,[r3+]
        addw    r13,MDL
        addcw   r12,MDH
        jmp     cc_nusr1,_2
    ; mac.c       18            sqr  += (long)b[i] * b[i];
    ; mac.c       19        }
    ; mac.c       20
    ; mac.c       21      *sum = dotp;
        movw    [r11],r13
        movw    [r11+#0x2],r12
    ; mac.c       22      return sqr;
        CoSTORE r2,MAL
        CoSTORE r3,MAH
    ; mac.c       23  }
        ret
```

Now 'sqr' has been allocated in the MAC accumulator. Because the lifetime of this object overlaps with that of 'dotp', the latter cannot be allocated in the accumulator.

1.14.3. MAC Support by Intrinsic Functions

It is also possible to use the MAC- unit by making use of intrinsic functions, which are described in Section 1.12.5, *Intrinsic Functions*. As is the case with manual qualification, the compiler will not automatically choose an object for MAC allocation when its lifetime overlaps with the intrinsic functions. It is possible to mix-in intrinsic functions though. Example:

```
#pragma mac
#pragma nosavemac
long f( int a, int b, int c, int d, long e )
{
    long sum;       /* __mac qualifier assigned automatically */

    sum  = (long)a * b;
    sum -= (long)c * d;
    /*
     * End lifetime of sum, start lifetime of "intrinsic functions"
     */
```

```
    __CoLOAD( sum );
    __CoNEG();
    __CoRND();
    /*
     * End lifetime of "intrinsic functions" start lifetime of sum
     */
    sum  = __CoSTORE() + e;
    sum <<= 1;
    sum += (long)a * d;
    return sum;
}
#pragma savemac
#pragma nomac
```

When compiled with **--core=xc16x**, this results in:

```
code_f  .section  code, new
        .global _f
_f      .proc   far
        CoMUL   r2,r3
        CoMACR  r4,r5,rnd
        CoADD   r11,r12
        CoSHL   #0x1
        CoMAC   r2,r5
        CoSTORE r2,MAL
        CoSTORE r3,MAH
        ret
```

From this example you can see that the __CoLOAD() and __CoSTORE() intrinsic functions are optimized away. The CoMAC generated by the C statement: sum -= (long)c * d; is combined with the __CoNEG() and __CoRND() intrinsic functions into a single instruction: CoMACR r4,r5,rnd.

1.14.4. Using the MAC Status Word

Just like the PSW flags, the MSW flags are compiler resources, they are not intended for direct use from the C code. If it is necessary to make decisions upon the MSW flags, use the intrinsics __CoSTOREMSW() or __CoCMP(). For example:

```
long f( void )
{
    __CoLOAD( x );
    if ( __CoCMP( y ) & (1 << 10) ) /* test the MC flag */
    {
        return x;
    }
    return y;
}
```

1.14.5. Evaluation of a Single Expression

When the MAC accumulator is not used by one of the previous methods, the code generator may decide to use the MAC unit for evaluation of a single (sub-) expression. The difference with (automatic) __mac qualification is that an object will not actually live inside the accumulator, but the accumulator will be temporarily used to evaluate a (complex) expression. Once the expression is evaluated, the result will be stored immediately into a register pair. This method is only useful when an expression is complex enough to compensate for the CoLOAD/CoSTORE operations needed to initialize/unload the accumulator.

Example:

```
#pragma mac
#pragma nosavemac
int * mac_shift( long value, int * buf )
{
    *buf++ = (int)(value >> 12);
    return buf;
}
#pragma savemac
#pragma nomac
```

When compiled with **--core=xc16x**, this will generate the following code:

```
  CoLOAD  r2,r3
  CoSHL   #0x4
  CoSTORE [r4+],MAH
  movw    r2,r4
  ret
```

Here, 'value' is temporarily loaded into the MAC accumulator, shifted, and the result is stored. The compiler can make use of the 40-bit shift instruction and the post-increment feature of the CoSTORE instruction.

1.14.6. Hardware Loops

When MAC instructions are generated in a loop body it is sometimes possible to convert a loop into a hardware loop. However, there are some prerequisites:

• There must be a MAC instruction on every path in the loop. This is necessary to ensure that the MRW register is updated in every iteration.

• The compiler must be able to find the iteration register, and the instruction that updates it. Furthermore, the update must match the update that is applied automatically to the MRW register when a MAC instruction is passed, i.e. subtract one in each iteration.

• The loop iteration variable cannot be used inside the loop or after the loop.

The compiler itself will actively assist by trying to transform a loop in such a form that it can be converted into a hardware loop. As you may have noticed in the previous examples, the loop iteration variable is used inside the loop and yet it is converted into a hardware loop. This is possible because the compiler

has applied strength reduction to the code. This optimization replaces the subscripted array by pointers. This eliminates the use of the loop variable inside the loop and enables the hardware loop optimization.

Loop optimizations are controlled by the option **-Ol** (loop transformations).

1.14.7. Considerations when Using the MAC

All MAC registers follow the callee-saves strategy. The costs for the save/restore of MAC registers in a function's prologue/epilogue are not taken into account during the cost analysis of the automatic allocation process. This is done on purpose, because the save/restore of MAC registers can sometimes be avoided. To do this, it is advised to pick the functions that you want to be MAC optimized with care. You can do this by enclosing a function in `#pragma mac/nomac` directives, rather than to enable the MAC unit application wide using the option **--mac**. Next, make sure that the MAC registers do not contain valid data when the functions you have picked are called. In general this will be the case when no caller or any other function up in the call tree that calls the MAC optimized function, uses MAC optimizations itself. It is now relatively easy to analyze this because the MAC unit is enabled selectively instead of application wide. When you are sure the MAC registers are not used, add `#pragma nosavemac/savemac` to disable the save/restore of MAC registers. By using these pragmas some significant overhead can be avoided.

1.15. Section Naming

The C compiler generates sections and uses a combination of the memory type and the object name as section names. The memory types are: code, near, far, huge, shuge, bit, bita and iram. See also Section 1.3.1, *Memory Type Qualifiers*. The section names are independent of the section attributes such as clear, init, and romdata.

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 rename sections with a pragma or with a command line option. The syntax is the same:

--rename-sections=[*type***=**]*format_string*[**,**[*type***=**]*format_string*]...

`#pragma section` [*type***=**]*format_string*[**,**[*type***=**]*format_string*]...

With the memory *type* you select which sections are renamed. The matching sections will get the specified format string for the 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
{type}	section type

The default compiler generated section names are {type}_{name}.

Some examples (file test.c):

```
#pragma section near={module}_{type}_{attrib}
__near int x;
/* Section name: test_near_near_clear */

#pragma section near=_c166_{module}_{name}
__near int status;
/* Section name: _c166_test_status */

#pragma section near=RENAMED_{name}
__near int barcode;
/* Section name: RENAMED_barcode */
```

With `#pragma endsection` the naming convention of the previous level is restored, while with `#pragma section default` 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`)

```
__near char a;         // allocated in 'near_a'
#pragma section near=MyNearData1
__near char b;         // allocated in 'MyNearData1'
#pragma section near=MyNearData2
__near char c;         // allocated in 'MyNearData2'
#pragma endsection
__near char d;         // allocated in 'MyNearData1'
#pragma endsection
__near char e;         // allocated in 'near_e'
```

1.16. TASKING Volatile Implementation

Volatile objects are objects defined with the keyword `volatile`. For example:

```
volatile unsigned int port;
```

Read and write operations on volatile variables in the C code always result in a read or write operation from/to the object. The compiler will not cache values of volatiles in registers. Operations on volatiles are kept in the order as they appear in the C code, even if there are no obvious relations between these operations. For non-volatile objects the compiler may change the order of operations when there is no dependency between these operations. This is also the case when intermixing volatile operations with non-volatile operations. The volatile operations will be kept in the sequence they have in the C code, but the non-volatile operations can be moved freely.

Mutual exclusion (mutex) can be used to avoid the simultaneous use of a common resource, such as a global variable, by pieces of C code called critical sections.

For example:

```
volatile unsigned int updating_data = 0;
unsigned long data;

void foo( unsigned long value )
```

```
{
    ...
    updating_data = 1;
    data = value; // The compiler can move this assignment
    updating_data = 0;
    ...
}
```

In this example an attempt is made to create a mutex with the volatile variable `updating_data`. But the compiler is still free to move 'critical code', the assignment of `data`, to earlier or later in the function, making the protection of the assignment void.

The C compiler has an option to make the volatile operations behave as fences for other operations, the **--language=+volatile** (**-Av**) option. In the given example the critical code can then never be moved across the setting and clearing of the mutex, nor is it possible that any other code moves into the critical code region.

It is quite common that when programming mutexes, a function call is used as a fence. For example:

```
void foo( unsigned long value )
{
    ...
    set_mutex( &updating_data );    // enter the critical code
    data = value;                   // do some safe processing
    clear_mutex( &updating_data );  // leave the critical code
    ...
}
```

In general this has the same effect as with the volatiles and the **--language=+volatile** (**-Av**) option. Even better, in this example the mutex `updating_data`, does not need to be volatile.

However, with advanced optimizations such as automatic inlining it is possible that the functions to set and clear the mutex will be inlined, meaning that the fences around the critical code have disappeared. Even putting the mutex set and clear functions into another module may not be safe when MIL linking is used in combination with automatic inlining. There are several solutions to avoid this problem:

• exclude the mutex set and clear functions from inlining, either by placing them in a module or library that is not included in the MIL linking or by adding the `__noinline` keyword to the function definitions and prototypes

• use volatile objects in combination with the **--language=+volatile** (**-Av**) option

• use fences with inline assembly, for example `__asm( "nop" );`

Chapter 2. C++ Language

The TASKING C++ compiler (**cp166**) offers a new approach to high-level language programming for the C166 family. The C++ compiler accepts the C++ language as defined by the ISO/IEC 14882:1998 standard and modified by TC1 for that standard. It also accepts the language extensions of the C compiler (see Chapter 1, *C Language*).

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

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

- The C++ Programming Language (second edition) by Bjarne Straustrup (1991, Addison Wesley)

- ISO/IEC 14882:1998 C++ standard [ANSI] More information on the standards can be found at http://www.ansi.org/

2.1. C++ Language Extension Keywords

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

Additionally the following language extensions are supported:

attributes

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

pragmas

The C++ compiler supports the same pragmas as the C compiler and some extra pragmas as explained in Section 2.8, *Pragmas to Control the C++ Compiler*. Pragmas give directions to the code generator of the compiler.

2.2. C++ Dialect Accepted

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

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

2.2.1. Standard Language Features Accepted

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

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

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

- Qualified names are allowed in elaborated type specifiers.

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

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

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

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

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

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

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

- Type template parameters are permitted to have default arguments.

- Function templates may have nontype template parameters.

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

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

- Digraphs are recognized.

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

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

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

- `bool` is recognized.

- RTTI (run-time type identification), including `dynamic_cast` and the `typeid` operator, is implemented.

- Declarations in tested conditions (in `if`, `switch`, `for`, and `while` statements) are supported.

- Array `new` and `delete` are implemented.

- New-style casts (`static_cast`, `reinterpret_cast`, and `const_cast`) are implemented.

- Definition of a nested class outside its enclosing class is allowed.

- `mutable` is accepted on non-static data member declarations.

- Namespaces are implemented, including `using` declarations and directives. Access declarations are broadened to match the corresponding `using` declarations.

- Explicit instantiation of templates is implemented.

- The `typename` keyword is recognized.

- `explicit` is accepted to declare non-converting constructors.

- The scope of a variable declared in the `for-init-statement` of a `for` loop is the scope of the loop (not the surrounding scope).

- Member templates are implemented.

- The new specialization syntax (using "`template <>`") is implemented.

- Cv-qualifiers are retained on rvalues (in particular, on function return values).

- The distinction between trivial and nontrivial constructors has been implemented, as has the distinction between PODs and non-PODs with trivial constructors.

- The linkage specification is treated as part of the function type (affecting function overloading and implicit conversions).

- `extern inline` functions are supported, and the default linkage for `inline` functions is external.

- A typedef name may be used in an explicit destructor call.

- Placement delete is implemented.

- An array allocated via a placement new can be deallocated via delete.

- Covariant return types on overriding virtual functions are supported.

- `enum` types are considered to be non-integral types.

- Partial specialization of class templates is implemented.

- Partial ordering of function templates is implemented.

- Function declarations that match a function template are regarded as independent functions, not as "guiding declarations" that are instances of the template.

- It is possible to overload operators using functions that take `enum` types and no `class` types.

- Explicit specification of function template arguments is supported.

- Unnamed template parameters are supported.

- The new lookup rules for member references of the form `x.A::B` and `p->A::B` are supported.

- The notation `:: template` (and `->template`, etc.) is supported.

- In a reference of the form `f()->g()`, with `g` a static member function, `f()` is evaluated. The ARM specifies that the left operand is not evaluated in such cases.

- `enum` types can contain values larger than can be contained in an `int`.

- Default arguments of function templates and member functions of class templates are instantiated only when the default argument is used in a call.

- String literals and wide string literals have `const` type.

- Class name injection is implemented.

- Argument-dependent (Koenig) lookup of function names is implemented.

- Class and function names declared only in unqualified friend declarations are not visible except for functions found by argument-dependent lookup.

- A `void` expression can be specified on a return statement in a `void` function.

- Function-try-blocks, i.e., try-blocks that are the top-level statements of functions, constructors, or destructors, are implemented.

- Universal character set escapes (e.g., `\uabcd`) are implemented.

- On a call in which the expression to the left of the opening parenthesis has class type, overload resolution looks for conversion functions that can convert the class object to pointer-to-function types, and each such pointed-to "surrogate function" type is evaluated alongside any other candidate functions.

- Dependent name lookup in templates is implemented. Nondependent names are looked up only in the context of the template definition. Dependent names are also looked up in the instantiation context, via argument-dependent lookup.

- Value-initialization is implemented. This form of initialization is indicated by an initializer of "()" and causes zeroing of certain POD-typed members, where the usual default-initialization would leave them uninitialized.

- A partial specialization of a class member template cannot be added outside of the class definition.

- Qualification conversions may be performed as part of the template argument deduction process.

- The `export` keyword for templates is implemented.

2.2.2. C++0x Language Features Accepted

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

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

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

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

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

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

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

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

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

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

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

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

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

2.2.3. Anachronisms Accepted

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.2.4. Extensions Accepted in Normal C++ Mode

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

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

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

- Constants of scalar type may be defined within classes:

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

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

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

- The `restrict` keyword is allowed.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.3. GNU Extensions

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

- Extended designators are accepted

- Compound literals are accepted.

- Non-standard anonymous unions are accepted

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

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

This can be useful in macro and template definitions.

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

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

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

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

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

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

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

- The C99 _Pragma operator is supported.

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

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

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

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

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

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

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

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

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

- Excess aggregate initializers are ignored with a warning.

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

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

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

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

- Extended variadic macros are supported.

- Dollar signs ($) are allowed in identifiers.

- Hexadecimal floating point constants are recognized.

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

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

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

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

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

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

- ASCII "NULL" characters are accepted in source files.

- A source file can end with a backslash ("\") character.

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

- A number of macros are predefined in GNU mode. See Section 2.9, *Predefined Macros*.

- A predefined macro can be undefined.

- A large number of special functions of the form `__builtin_xyz` (e.g., `__builtin_alloca`) are predeclared.

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

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

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

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

- Trigraphs are ignored (with a warning).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
typedef void F();

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

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

- The macro _GNU_SOURCE is defined as "1".

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

- Namespace std is predeclared.

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

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

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

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

```
extern "C" int f(void);
extern "C" int g(void);
namespace N {
    extern "C" int f(void);  // same type
    extern "C" void g(void); // different type
```

```
};
using namespace N;
int i = f(); // calls ::f
int j = g(); // calls ::f
```

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

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

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

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

- A `friend` declaration may refer to a member typedef.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- When gnu_version is < 30400, the semantic analysis of a friend function defined in a class template is performed only if the function is actually used and is done at the end of the translation unit (instead of at the point of first use).

- A function template default argument may be redeclared. A warning is issued and the default from the initial declaration is used.

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

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

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

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

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

- Default arguments are retained as part of deduced function types.

- A namespace member may be redeclared outside of its namespace.

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

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

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

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

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

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

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

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

- Strong using-directives are supported.

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

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

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

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

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

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

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

```
    A& a;
};
```

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

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

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

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

The following GNU extensions are *not* currently supported:

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

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

- Nested functions

2.4. Namespace Support

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

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

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

Dependent Name Processing

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

Lookup Using the Referencing Context

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

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

```
namespace N {
    int g(int);
    int x = 0;
    template <class T> struct A {
        T f(T t) { return g(t); }
        T f() { return x; }
    };
}

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

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

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

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

Argument Dependent Lookup

When argument-dependent lookup is enabled (this is the default), functions made visible using argument-dependent lookup overload with those made visible by normal lookup. The standard requires that this overloading occurs even when the name found by normal lookup is a block extern declaration. The C++ compiler does this overloading, but in default mode, argument-dependent lookup is suppressed when the normal lookup finds a block extern.

This means a program can have different behavior, depending on whether it is compiled with or without argument-dependent lookup **--no-arg-dep-lookup**, even if the program makes no use of namespaces. For example:

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

2.5. Template Instantiation

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

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

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

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

---

[1] Since templates are descriptions of entities (typically, classes) that are parameterizable according to the types they operate upon, they are sometimes called *parameterized types*.

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

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

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

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

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

2.5.1. Automatic Instantiation

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

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

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

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

  This scheme has the disadvantage that it does a separate compilation for each instantiated function (or, at best, one compilation for all the member functions of one class). Even though the function itself is often quite small, it must be compiled along with the declarations for the types on which the instantiation is based, and those declarations can easily run into many thousands of lines. For large systems, these compilations can take a very long time. The link step tries to be smart about recompiling instantiations only when necessary, but because it keeps no fine-grained dependency information, it is often forced

---

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

to "recompile the world" for a minor change in a `.h` file. In addition, *cfront* has no way of ensuring that preprocessing symbols are set correctly when it does these instantiation compilations, if preprocessing symbols are set other than on the command line.

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

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

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

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

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

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

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

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

2.5.2. Instantiation Modes

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

--instantiate=used

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

---

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

--instantiate=all

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

--instantiate=local

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

In the case where the **cc166** command is given a single file to compile and link, e.g.,

```
cc166 test.cc
```

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

2.5.3. Instantiation #pragma Directives

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

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

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

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

The argument to the instantiation pragma may be:

• a template class name `A<int>`

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

• a member function name `A<int>::f`

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

• a static data declaration `int A<int>::i`

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

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

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

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

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

```
template <class T> void f1(T);  // No body provided
template <class T> void g1(T);  // No body provided

void f1(int) {} // Specific definition
void main()
{
    int i;
    double d;
    f1(i);
    f1(d);
    g1(i);
    g1(d);
}

#pragma instantiate void f1(int) // error - specific
                                 // definition
#pragma instantiate void g1(int) // error - no body
                                 // provided
```

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

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

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

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

2.5.4. Implicit Inclusion

When implicit inclusion is enabled, the C++ compiler is given permission to assume that if it needs a definition to instantiate a template entity declared in a `.h` file it can implicitly include the corresponding `.cc` file to get the source code for the definition. For example, if a template entity `ABC::f` is declared in file `xyz.h`, and an instantiation of `ABC::f` is required in a compilation but no definition of `ABC::f` appears

in the source code processed by the compilation, the compiler will look to see if a file `xyz.cc` exists, and if so it will process it as if it were included at the end of the main source file.

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

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

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

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

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

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

2.5.5. Exported Templates

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

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

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

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

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

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

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

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

2.5.5.1. Finding the Exported Template Definition

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

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

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

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

2.5.5.2. Secondary Translation Units

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

instantiations of exported templates, which in turn can cause additional secondary translation units to be loaded[4].

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

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

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

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

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

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

2.5.5.3. Libraries with Exported Templates

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

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

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

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

2. Concatenate all of the `.et` files into a single `.et` file (e.g., `mylib.et`) in the `export` directory. The individual `.et` files could be copied to the export directory, but having all of the `.et` information in one file will make use of the library more efficient.

---

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

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

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

```
include=x
```

or

```
sys_include=x
```

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

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

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

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

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

2.6. Inlining Functions

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

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

• The function has local static variables.

• The function has local constants.

- The function has local types.

- The function has block scopes.

- The function includes pragmas.

- The function has a variable argument list.

2.7. Extern Inline Functions

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

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

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

2.8. Pragmas to Control the C++ Compiler

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

The syntax is:

```
#pragma pragma-spec
```

The C++ compiler supports the following pragmas and all C compiler pragmas that are described in Section 1.6, *Pragmas to Control the Compiler*

instantiate / do_not_instantiate / can_instantiate

These are template instantiation pragmas. They are described in detail in Section 2.5.3, *Instantiation #pragma Directives*.

hdrstop / no_pch

These are precompiled header pragmas. They are described in detail in Section 2.10, *Precompiled Headers*.

once

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

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

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

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

ident

This pragma is given in the form:

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

or

```
#ident "string"
```

2.9. Predefined Macros

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

Macro	Description
__ABI_COMPATIBILITY_VERSION	Defines the ABI compatibility version being used. This macro is set to 9999, which means the latest version. This macro is used when building the C++ library.
__ABI_CHANGES_FOR_RTTI	This macro is set to TRUE, meaning that the ABI changes for RTTI are implemented. This macro is used when building the C++ library.
__ABI_CHANGES_FOR_ARRAY_NEW_AND_DELETE	This macro is set to TRUE, meaning that the ABI changes for array new and delete are implemented. This macro is used when building the C++ library.

Macro	Description
__ABI_CHANGES_FOR_PLACEMENT_DELETE	This macro is set to TRUE, meaning that the ABI changes for placement delete are implemented. This macro is used when building the C++ library.
__ARRAY_OPERATORS	Defined when array `new` and `delete` are enabled. This is the default.
__BASE_FILE__	Similar to __FILE__ but indicates the primary source file rather than the current one (i.e., when the current file is an included file).
_BOOL	Defined when `bool` is a keyword. This is the default.
__BUILD__	Identifies the build number of the C++ compiler, 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.
__CHAR_MIN / __CHAR_MAX	Used in `limits.h` to define the minimum/maximum value of a plain `char` respectively.
__CP166__	Identifies the C++ compiler. You can use this symbol to flag parts of the source which must be recognized by the **cp166** C++ compiler only. It expands to 1.
__CORE__	Expands to a string with the core depending on the C++ compiler options **--cpu** and **--core**. The symbol expands to "c16x" when no **--cpu** and no **--core** is supplied.
__cplusplus	Always defined.
__CPU__	Expands to a string with the CPU supplied with the option **--cpu**. When no **--cpu** is supplied, this symbol is not defined.
__DATE__	Defined to the date of the compilation in the form "Mmm dd yyyy".
__DELTA_TYPE	Defines the type of the offset field in the virtual function table. This macro is used when building the C++ library.
__DOUBLE_FP__	Expands to 1 if you did *not* use option **--no-double** (Treat 'double' as 'float'), otherwise unrecognized as macro.

Macro	Description
__embedded_cplusplus	Defined as 1 in Embedded C++ mode.
__EXCEPTIONS	Defined when exception handling is enabled (**--exceptions**).
__FILE__	Expands to the current source file name.
__FUNCTION__	Defined to the name of the current function. An error is issued if it is used outside of a function.
__func__	Same as __FUNCTION__ in GNU mode.
__IMPLICIT_USING_STD	Defined when the standard header files should implicitly do a using-directive on the std namespace (**--using-std**).
__JMP_BUF_ELEMENT_TYPE	Specifies the type of an element of the setjmp buffer. This macro is used when building the C++ library.
__JMP_BUF_NUM_ELEMENTS	Defines the number of elements in the setjmp buffer. This macro is used when building the C++ library.
__LINE__	Expands to the line number of the line where this macro is called.
__MODEL__	Identifies the memory model for which the current module is compiled. It expands to a single character constant: 'n' (near), 'f' (far), 's' (shuge) or 'h' (huge).
__NAMESPACES	Defined when namespaces are supported (this is the default, you can disable support for namespaces with **--no-namespaces**).
__NO_LONG_LONG	Defined when the long long type is not supported. This is the default.
__NULL_EH_REGION_NUMBER	Defines the value used as the null region number value in the exception handling tables. This macro is used when building the C++ library.
__PLACEMENT_DELETE	Defined when placement delete is enabled.
__PRETTY_FUNCTION__	Defined to the name of the current function. This includes the return type and parameter types of the function. An error is issued if it is used outside of a function.
__PTRDIFF_MIN / __PTRDIFF_MAX	Used in stdint.h to define the minimum/maximum value of a ptrdiff_t type respectively.

Macro	Description
__REGION_NUMBER_TYPE	Defines the type of a region number field in the exception handling tables. This macro is used when building the C++ library.
__REVISION__	Expands to the revision number of the C++ compiler. Digits are represented as they are; characters (for prototypes, alphas, betas) are represented by -1. Examples: v1.0r1 -> 1, v1.0rb -> -1
__RTTI	Defined when RTTI is enabled (**--rtti**).
__RUNTIME_USES_NAMESPACES	Defined when the run-time uses namespaces.
__SFRFILE__(*cpu*)	This macro expands to the filename of the used SFR file, including the pathname and the < >. The *cpu* is the argument of the macro. For example, if **--cpu=xc167ci** is specified, the macro `__SFRFILE__(__CPU__)` expands to `__SFRFILE__(xc167ci)`, which expands to `<sfr/regxc167ci.sfr>`.
__SIGNED_CHARS__	Defined when plain `char` is signed.
__SINGLE_FP__	Expands to 1 if you used option **--no-double** (Treat 'double' as 'float'), otherwise unrecognized as macro.
__SIZE_MIN / __SIZE_MAX	Used in `stdint.h` to define the minimum/maximum value of a `size_t` type respectively.
__STDC__	Always defined, but the value may be redefined.
__STDC_VERSION__	Identifies the ISO-C version number. Expands to 199901L for ISO C99, but the value may be redefined.
_STLP_NO_IOSTREAMS	Defined when option **--io-streams** is not used. This disables I/O stream functions in the STLport C++ library.
__TASKING__	Always defined for the TASKING C++ compiler.
__TIME__	Expands to the compilation time: "hh:mm:ss"
__TYPE_TRAITS_ENABLED	Defined when type traits pseudo-functions (to ease the implementation of ISO/IEC TR 19768; e.g., `__is_union`) are enabled. This is the default in C++ mode.
__VAR_HANDLE_TYPE	Defines the type of the variable-handle field in the exception handling tables. This macro is used when building the C++ library.

Macro	Description
__VERSION__	Identifies the version number of the C++ compiler. For example, if you use version 2.1r1 of the compiler, __VERSION__ expands to 2001 (dot and revision number are omitted, minor version number in 3 digits).
__VIRTUAL_FUNCTION_INDEX_TYPE	Defines the type of the virtual function index field of the virtual function table. This macro is used when building the C++ library.
__VIRTUAL_FUNCTION_TYPE	Defines the type of the virtual function field of the virtual function table. This macro is used when building the C++ library.
__WCHAR_MIN / __WCHAR_MAX	Used in stdint.h to define the minimum/maximum value of a wchar_t type respectively.
_WCHAR_T	Defined when wchar_t is a keyword.

2.10. Precompiled Headers

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

2.10.1. Automatic Precompiled Header Processing

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

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

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

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

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

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

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

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

  ```
  // xxx.h
  class A {

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

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

  ```
  // yyy.h
  static

  // yyy.C
  #include "yyy.h"
  int i;
  ```

- Similarly, the header stop point may not be inside a `#if` block or a `#define` started within a header file.

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

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

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

- **#pragma no_pch** (see below) must not have appeared.

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

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

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

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

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

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

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

- The command line options.

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

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

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

As an illustration, consider two source files:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.10.2. Manual Precompiled Header Processing

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

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

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

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

2.10.3. Other Ways to Control Precompiled Headers

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

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

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

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

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

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

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

2.10.4. Performance Issues

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

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

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

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

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

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

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

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

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

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

Chapter 3. Assembly Language

This chapter describes the most important aspects of the TASKING assembly language. For a complete overview of the architecture you are using, refer to the target's Core Reference Manual.

3.1. Assembly Syntax

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

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

The syntax of an assembly statement is:

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

label

A label is a special symbol which is assigned the value and type of the current program location counter. A label can consist of letters, digits, dollar ($) and underscore characters (_). The first character cannot be a digit or a $. 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 3.3, *Operands of an Assembly Instruction*. The instructions are described in the target's Core Reference Manual.
	The instruction can also be a so-called 'generic instruction'. Generic instructions are pseudo instructions (no instructions from the instruction set). Depending on the situation in which a generic instruction is used, the assembler replaces the generic instruction with appropriate real assembly instruction(s). For a complete list, see Section 3.11, *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 3.9, *Assembler Directives and Controls*.
macro_call	A call to a previously defined macro. It must not start in the first column. See Section 3.10, *Macro Operations*.
comment	Comment, preceded by a ; (semicolon).

You can use empty lines or lines with only comments.

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

$*control*

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

3.2. Assembler Significant Characters

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

Some characters have a special meaning to the assembler. Special characters associated with expression evaluation are described in Section 3.7.3, *Expression Operators*. Other special assembler characters are:

Character	Description
;	Start of a comment
;;	Unreported comment delimiter
\	Line continuation character or macro operator: argument concatenation
?	Macro operator: return decimal value of a symbol
%	Macro operator: return hex value of a symbol
^	Macro operator: override local label
"	Macro string delimiter or quoted string `.DEFINE` expansion character
'	String constants delimiter

Character	Description
@	Start of a built-in assembly function
$	Location counter substitution
[ ]	Instruction grouping operator
#	Immediate addressing

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

3.3. Operands of an Assembly Instruction

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

Operand	Description
symbol	A symbolic name as described in Section 3.4, *Symbol Names*. Symbols can also occur in expressions.
register	Any valid register as listed in Section 3.5, *Registers*.
expression	Any valid expression as described in Section 3.7, *Assembly Expressions*.
address	A combination of *expression*, *register* and *symbol*.

3.4. Symbol Names

User-defined symbols

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

Predefined preprocessor symbols

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

Labels

Symbols used for memory locations are referred to as labels.

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
```

3.4.1. Predefined Preprocessor Symbols

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

Symbol	Description
__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.
__C166__	Identifies the assembler. You can use this symbol to flag parts of the source which must be recognized by the **as166** assembler only. It expands to 1.
__CORE__	Expands to a string with the core depending on the assembler options **--cpu** and **--core**. The symbol expands to "c16x" when no **--cpu** and no **--core** is supplied.
__CORE_*core*__	A symbol is defined depending on the options **--cpu** and **--core**. The *core* is converted to upper case. Example: if **--cpu=xc167ci** is specified, the symbol __CORE_XC16X__ is defined. When no **--core** or **--cpu** is supplied, the assembler defines __CORE_C16X__.
__CPU__	Expands to a string with the CPU supplied with the option **--cpu**. When no **--cpu** is supplied, this symbol is not defined.
__CPU_*cpu*__	A symbol is defined depending on the option **--cpu=*cpu*. The *cpu* is converted to uppercase. For example, if **--cpu=xc167ci** is specified the symbol __CPU_XC167CI__ is defined. When no **--cpu** is supplied, this symbol is not defined.
__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
__SILICON_BUG_*num*__	This symbol is defined if the number *num* is defined with the option **--silicon-bug**.

Symbol	Description
__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('__CPU_XC167CI__')
  ; this part is only for the XC167CI
...
.endif
```

3.5. Registers

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

```
R0  .. R15   (general purpose registers)
RL0 .. RL7   (byte registers)
RH0 .. RH7   (byte registers)
```

3.6. Special Function Registers

It is easy to access Special Function Registers (SFRs) that relate to peripherals from assembly. The SFRs are defined in a special function register file (*.sfr) as symbol names for use with the compiler and assembler. The assembler reads the SFR file as defined by the selected derivative with the command line option **--cpu (-C)**. The format of the SFR file is exactly the same as the include file for the C compiler. For more details on the SFR files see Section 1.3.5, *Accessing Hardware from C*. Because the SFR file format uses C syntax and the assembler has a limited C parser, it is important that you only use the described constructs.

SFRs in the SFR area and extended SFR area are addressed in the near address space. Some SFRs cannot be addressed with a REG addressing mode, although they are within the SFR area or the extended SFR area. These registers are:

RSTCON	0xF1E0
RSTCON2	0xF1E2
SYSSTAT	0xF1E4

Example use in assembly:

```
 movw P0L,#0x88  ; use of port name
 bset P0L_3      ; use of bit name
 jnb  P0L_4,_2
 bclr P0L_3
```

```
_2:
 bset IEN         ; use of bit name
```

Without an SFR file the assembler only knows the general purpose registers R0-R15 and the SFRs PSW (and its bits), DPP0, DPP1, DPP2 and DPP3.

3.7. Assembly Expressions

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

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

Expressions follow the conventional rules of algebra and boolean arithmetic.

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

When any operand of an expression is relocatable, the entire expression is relocatable. Relocatable expressions are emitted in the object file and evaluated by the linker.

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

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

*   *numeric constant*

*   *string*

*   *symbol*

*   *expression binary_operator expression*

*   *unary_operator expression*

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

*   *function call*

All types of expressions are explained in separate sections.

3.7.1. 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 and suffixes 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** or **y** suffix	`0b1101` `11001010b`

Base	Description	Example
Octal	Octal digits (0-7) followed by a **o** suffix	`777o`

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

Decimal	Decimal digits (0-9), optionally followed by a **d** or **t**	`12` `1245d`

3.7.2. Strings

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

Strings constants in expressions are evaluated to a number (each character is replaced by its ASCII value). Strings in expressions can have a size of up to 8 characters or less depending on the operand of an instruction or directive; any subsequent characters in the string are ignored. In this case the assembler issues a warning. An exception to this rule is when a string is used in a `.DB` assembler directive; in that case all characters result in a constant value of the specified size. Null strings have a value of 0.

Examples

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

3.7.3. Expression Operators

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

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

Type	Operator	Name	Description
	( )	parenthesis	Expressions enclosed by parenthesis are evaluated first.
Unary	+	plus	Returns the value of its operand.
	-	minus	Returns the negative of its operand.
	~	one's complement	Integer only. Returns the one's complement of its operand. It cannot be used with a floating-point operand.

Type	Operator	Name	Description
	! NOT	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.
	DPP*n*:	DPP override	Specify the DPP number used in bit 14 and 15 of the address. The DPP*n* is one of DPP0, DPP1, DPP2, DPP3
	PAG	page number	Returns the page number of the operand (operand >> 14), same as @pag() function.
	POF	page offset	Returns the page offset of the operand (operand & 0x3FFF), same as @pof() function.
	SEG	segment number	Returns the segment number of the operand (operand >> 16), same as @seg() function.
	SOF	segment offset	Returns the segment offset of the operand (operand & 0xFFFF), same as @sof() function.
	BOF	bit offset	Returns the bit offset of a bit operand, same as @bof() function.
	HIGH	high byte	Returns the high byte of the operand ((operand >> 8)&0xFF), same as @msb() function.
	LOW	low byte	Returns the low byte of the operand (operand & 0xFF), same as @lsb() function.
	type	type cast	Any of the valid assembler symbol types can be used as a type cast operator.
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.
	% MOD	modulo	Used with integers, this operator yields the remainder from the division of the first operand by the second. Used with floating-point operands, this operator applies the following rules: Y % Z = Y if Z = 0 Y % Z = X if Z <> 0, where X has the same sign as Y, is less than Z, and satisfies the relationship: Y = integer * Z + X
	+	addition	Yields the sum of its operands.
	-	subtraction	Yields the difference of its operands.
Shift	<< SHL	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.

Type	Operator	Name	Description
	>> SHR	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	< LT	less than	Returns an integer 1 if the indicated condition is TRUE or an integer 0 if the indicated condition is FALSE.

In either case, the memory space attribute of the result is N

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. |
	<= LE	less than or equal	
	> GT	greater than	
	>= GE	greater than or equal	
	== EQ	equal	
	!= NE	not equal	
	ULT	unsigned less than	The unsigned operators are implemented as signed operators that mask out the top bit of the expressions. This makes them effectively 63-bit operators.
	ULE	unsigned less than or equal	
	UGT	unsigned greater than	
	UGE	unsigned greater than or equal	
Bit and Bitwise	.	bit position	Specify bit position (right operand) in a bit addressable byte or word (left operand).
	& AND	AND	Integer only. Yields the bitwise AND function of its operand.
	| OR	OR	Integer only. Yields the bitwise OR function of its operand.
	^ XOR	exclusive OR	Integer only. Yields the bitwise exclusive OR function of its operands.
Logical	&&	logical AND	Returns an integer 1 if both operands are non-zero; otherwise, it returns an integer 0.
	||	logical OR	Returns an integer 1 if either of the operands is non-zero; otherwise, it returns an integer 1

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

3.7.4. Symbol Types and Expression Types

Symbol Types

The type of a symbol is determined upon its definition by the directive it is defined with and by the section in which it is defined. The following table shows the symbol types that are available.

Symbol type	Section type where symbol is defined	Directive resulting in the symbol type
NEAR	CODE	with or after a `.PROC NEAR`
FAR	CODE	with or after a `.PROC FAR`
BIT	BIT	.dbit, .dsbit, .ds, .bit
BYTE	FAR, SHUGE, HUGE	.db, .ds, .dsb
WORD	FAR, SHUGE, HUGE	.dw, .dl, .dll, dsw, dsl, dsll
BITBYTE	BIT, BITA	.db, .dsb
BITWORD	BIT, BITA	.dw, .dl, .dll, dsw, dsl, dsll
NEARBYTE	NEAR, IRAM	.db, .ds, .dsb
NEARWORD	NEAR, IRAM	.dw, .dl, .dll, dsw, dsl, dsll
DATA3		.equ, .set
DATA4		.equ, .set
DATA8		.equ, .set
DATA16		.equ, .set
INTNO	CODE	.proc intno
REGBANK		.regbank, .label
SFR		.extern (internal)

Besides the mentioned directives it is also possible to explicitly define the symbol's type with the `.LABEL` directive and with the `.EXTERN` directive. Labels not on the same line as the directive still are assigned the type for that directive if they immediately precede the directive:

```
farsect .section far
mylabel:  ; this label gets the WORD type
 .dw 1
```

When you make a symbol global with the `.GLOBAL` directive, the symbol's type will be stored in the object file. The `.EXTERN` directive used for importing the symbol in another module must specify the same type. If the type is omitted in the `.EXTERN` directive, the assembler will assume the following when using the symbol:

Symbol used in	Symbol type
bit operation	BIT
byte operation	BYTE

Symbol used in	Symbol type
word operation	WORD
left of dot operator	BITWORD
generic call	FAR
immediate operands	DATA16

If none of the directives are used that result in a symbol type, the symbol gets a default type based on the section it is defined in:

Section type	Default symbol type	Possible symbol types
BIT	BIT	BIT
BITA	BITWORD	BITWORD, BITBYTE
IRAM	NEARWORD	NEARWORD, NEARBYTE, REGBANK
NEAR	NEARWORD	NEARWORD, NEARBYTE
FAR	WORD	WORD, BYTE
SHUGE	WORD	WORD, BYTE
HUGE	WORD	WORD, BYTE
CODE	FAR	FAR, NEAR, INTNO

For creating bit addressable bytes or words with the type BITBYTE or BITWORD, BIT or BITA sections must be used. For defining a BITBYTE the label must be byte aligned and for a BITWORD it must be word aligned.

Example with a BITA section:

```
bitasect .section bita
bitb     .dsb   1    ; BITBYTE
         .align 2
bitw     .dsw   1    ; BITWORD
```

The .ALIGN directive is used here because the assembler issues a warning on unaligned word definitions.

Symbols defined with .EQU or .SET inherit the type of the expression. The result of an expression is determined by the type of symbols used in the expression.

Symbols of type WORD or BYTE

As you can see from the table above, the assembler cannot make a difference between a far, shuge and a huge symbol, it only knows the symbol types WORD and BYTE as possible symbol types in a far, shuge or huge section. As a consequence the linker also cannot know whether the symbol is far, shuge or huge. This can result in an error from the linker, E109: address space mismatch. For example, when a huge symbol is located in shuge memory. To workaround this, use a section_layout::huge in the LSL file to assign a value to a far or shuge symbol.

Type Checking

When you use a symbol or expression as an operand for an instruction, the assembler will check if the type of this symbol or expression is valid for the used instruction. If it is not valid, the assembler will issue an error. For generic instructions the assembler uses the symbol type to select the smallest instruction.

When a relocatable expression is used as a word address operand, the linker checks if the result of the expression is word aligned. An error will be issued if this is not the case. This is done independently of the used type.

Expression Types

When evaluating an expression, the result of the expression is determined by the operands of the expression and the operators. The types of the symbols are divided in two groups: constant types and address types

| Constant types: | DATA3, DATA4, DATA8, DATA16 and INTNO |
| Address types: | NEAR, FAR, BIT, BYTE, WORD, BITBYTE, BITWORD, NEARBYTE, NEARWORD and REGBANK |

Address types may each relate to incompatible memory spaces. Unary operators are not allowed on address types. A unary operator applied to a constant type will yield the same constant type as result of the expression. The following table shows the resulting operand types for a binary operator:

Binary operator	Operand combination		
	Constant/Constant	Address/Constant or Constant/Address	Address/Address
- (subtraction)	Largest constant type	Address type remarks: the section information of the address operand is used for the result	Constant type if the address types are compatible. Illegal address operation if the addresses are incompatible. remarks: There is no relocation if both addresses are from the same section.
bitwise OR, XOR and AND	Largest constant type	Address type	Address type Illegal address operation if the addresses are incompatible. remarks: There is no relocation if both addresses are from the same section.

Binary operator	Operand combination		
	Constant/Constant	**Address/Constant or Constant/Address**	**Address/Address**
+ (addition)	Largest constant type	Address type	Address type Illegal address operation if the addresses are incompatible. remarks: There is no relocation if both addresses are from the same section.
. (dot)	BIT	BIT remarks: only allowed if type of address is BITBYTE or BITWORD	Illegal address operation
==, EQ, !=, NE, >=, GE, <=, LE, >, GT, <, LT, ULT, UGT, ULE, UGE	DATA3		
other binary operator	Largest constant type	Address type remarks: the section information of the address operand is used for the result	Illegal address operation

3.8. Built-in Assembly Functions

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

Syntax of an assembly function

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

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

The names of assembly functions are case insensitive.

Overview of assembly functions

Function	Description
@ABS(*expr*)	Absolute value
@ARG('*symbol*' | *expr*)	Test whether macro argument is present
@BITBYTE(*expr*)	Bitbyte of the expression
@BITWORD(*expr*)	Bitword of the expression
@BOF(*expr*)	Bit offset of the expression
@CNT()	Return number of macro arguments
@DEFINED('*symbol*' | *symbol*)	Test whether *symbol* exists
@DPP(*label*)	Return DPP register to access the label
@FAR(*expr*)	Far result of the expression
@LSB(*expr*)	Least significant byte of the expression
@LSW(*expr*)	Least significant word of the expression
@MSB(*expr*)	Most significant byte of the expression
@MSW(*expr*)	Most significant word of the expression
@NEAR(*expr*)	Near result of the expression
@PAG(*expr*)	Page number of the expression
@POF(*expr*)	Page offset of the expression
@SEG(*expr*)	Segment number of the expression
@SOF(*expr*)	Segment offset of the expression
@STRCAT(*str1*,*str2*)	Concatenate *str1* and *str2*
@STRCMP(*str1*,*str2*)	Compare *str1* with *str2*
@STRCMPI(*str1*,*str2*)	Compare *str1* with *str2* case insensitive
@STRLEN(*string*)	Return length of *string*
@STRPOS(*str1*,*str2*[,*start*])	Return position of *str2* in *str1*
@SUBSTR(*str*,*expr1*,*expr2*)	Return substring

Detailed Description of Built-in Assembly Functions

@ABS(*expression*)

Returns the absolute value of the expression.

Example:

```
AVAL  .SET @ABS(-2)   ; AVAL = 2
```

@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?
```

@BITBYTE(*expression*)

Returns the bitbyte of the result of the expression. The result of the expression must be a bit address.

@BITWORD(*expression*)

Returns the bitword of the result of the expression. The result of the expression must be a bit address.

@BOF(*expression*)

Returns the bit offset of the result of the expression. The result of the expression must be a bit address

@CNT()

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

Example:

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

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

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

Example:

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

@DPP(*label*)

Expands to the DPP register needed to access the near *label*. The assembler issues an error if the *label* is not of the type near. Function can be used anywhere where a short or long address can be used, including expressions.

@FAR(*expression*)

Returns the far result of the expression.

@LSB(*expression*)

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

Example:

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

@LSW(*expression*)

Returns the least significant word of the result of the expression. The result of the expression is calculated as a long (32 bit).

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 bit.

@MSW(*expression*)

Returns the most significant word of the result of the expression. The result of the expression is calculated as a long (32 bit).

@NEAR(*expression*)

Returns the near result of the expression.

@PAG(*expression*)

Returns the page number of the result of the expression. The result of the expression is calculated as long (32 bit).

Example:

```
ISEC  .SECTION near,init
AWORD .DW @PAG(COUNT)  ; Initialize with the page number where COUNT is located.
COUNT .DS 1
ISEC  .ENDS
```

@POF(*expression*)

Returns the page offset of the result of the expression. The result of the expression is calculated as long (32 bit).

Example:

```
DSEC  .SECTION near,init
TAB2  .DW 8
DSEC  .ENDS

CSEC  .SECTION code
      MOV R0, #@POF(TAB2) ; Fill R0 with the page offset
                          ; offset of variable TAB2
CSEC  .ENDS
```

@SEG(*expression*)

Returns the segment number of the result of the expression. The result of the expression is calculated as long (32 bit).

Example:

```
DSEC .SECTION near,init
AWORD .DW @SEG(TABX)  ; Initialize with the segment number where TABX is located.
TABX .DS 1
TABY .DS 20
DSEC .ENDS
```

@SOF(*expression*)

Returns the segment offset of the result of the expression. The result of the expression is calculated as long (32 bit).

@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'?
```

@STRCMPI(*string1*,*string2*)

Same as `@STRCMP()`, but compares strings case insensitive.

@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
```

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

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

Example:

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

3.9. Assembler Directives and Controls

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

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

   The following directives fall under this group:

   • Assembly control directives

   • Symbol definition directives

   • Data definition / Storage allocation directives

   • High Level Language (HLL) directives

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

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

   The following controls are available:

   • Assembly listing controls

   • Miscellaneous controls

Each assembler directive or control has its own syntax. You can use assembler directives and controls in the assembly code as pseudo instructions. The assembler recognizes both upper and lower case for directives and controls.

3.9.1. Assembler Directives

Overview of assembly control directives

Directive	Description
.END	Indicates the end of an assembly module
.INCLUDE	Include file

Overview of symbol definition directives

Directive	Description
.ALIAS	Create an alias for a symbol
.ASSUME	Assume DPP usage
.CGROUP, .DGROUP	Create a group of code sections or data sections
.EQU	Set permanent value to a symbol
.EXTERN	Import global section symbol
.GLOBAL	Declare global section symbol
.LABEL	Define a label of a specified type
.PROC, .ENDP	Define a procedure
.REGBANK	Define register bank
.SECTION, .ENDS	Start a new section
.SET	Set temporary value to a symbol
.SOURCE	Specify name of original C source file
.WEAK	Mark a symbol as 'weak'

Overview of data definition / storage allocation directives

Directive	Description
.ALIGN	Align location counter
.DBIT	Define bit
.DB	Define byte
.DW	Define word (16 bits)
.DL	Define long (32 bits)
.DLL	Define long long (64 bits)
.DBFILL, .DWFILL, .DLFILL, .DLLFILL	Fill block of memory
.DBPTR, .DPPTR, .DSPTR	Define pointer values in memory
.DS, .DSBIT, .DSW, .DSL, .DSLL	Define storage

Overview of macro and conditional assembly 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

Directive	Description
`.IF`, `.ELIF`, `.ELSE`	Conditional assembly directive
`.ENDIF`	End of conditional assembly directive
`.MACRO`, `.ENDM`	Define macro
`.UNDEF`	Undefine `.DEFINE` symbol or macro

Overview of HLL directives

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

.ALIAS

Syntax

alias-name **.ALIAS** *function-name*

Description

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

Example

_malloc .ALIAS ___hmalloc

.ALIGN

Syntax

.ALIGN *expression*

Description

With the .ALIGN directive you instruct the assembler to align the location counter. By default the assembler aligns on the alignment specified with the .SECTION directive.

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

```
CSEC  .section code
  .ALIGN 16    ; the assembler aligns
  instruction  ; this instruction at 16 MAUs and
               ; fills the 'gap' with NOP instructions.

CSEC2 .section code
  .ALIGN 12    ; WRONG: not a power of two, the
  instruction  ; assembler aligns this instruction at
               ; 16 MAUs and issues a warning.
```

Example with a BIT section to create a bit addressable byte or word with the type BITBYTE or BITWORD:

```
bitsect .section bit,word
        .ds 1    ; single bit
        .ALIGN 8
bb      .dsb 1   ; BITBYTE
        .ALIGN 16
bw      .dsw 1   ; BITWORD
```

The section is word aligned, because of the .dsw directive. It is impossible to align the .dsw directive correctly if the section is not aligned at word or a multiple of words. The .ALIGN directives are needed to place the .dsb and .dsw directives at the correct location.

.ASSUME

Syntax

.ASSUME DPP*n***:***sectpart*[**, DPP***n***:***sectpart*]...

or

.ASSUME NOTHING

Description

You can use the `.ASSUME` directive to specify what the contents of the DPP registers will be at run-time. This is done to help the assembler to ensure that the data referenced will be addressable.

The assembler checks each data memory reference for addressability based on the contents of the `.ASSUME` directive. The `.ASSUME` directive does not initialize the DPP registers; it is used by the assembler to help you be aware of the addressability of the data. Unless the data is addressable (as defined either by an `.ASSUME` or a page override), the assembler produces an error.

Field values

DPP*n*

One of the C166 Data Page Pointer (DPP) registers: DPP0, DPP1, DPP2, and DPP3.

sectpart

With this field you can define a page number. It can have the following values:

- section name, as in `.ASSUME DPP0:DSEC1, DPP1:DSEC3`

  All variables and labels defined in section DSEC1 are addressed with DPP0 and all variables defined in the section DSEC3 are addressed with DPP1. This applies to all sections with the same name in the current module.

- group name, as in `.ASSUME DPP2:DGRP`

  All variables and labels defined in sections which are member of the group DGRP are addressed with DPP2.

- variable name or label name, as in `.ASSUME DPP0:VarOrLabName`

  If the variable or label name is defined in a module internal section, all variables or labels defined in this section are addressed with DPP0. If the variable or label name is defined in a module-external section, only this variable can be addressed with DPP0.

- `NOTHING` keyword, as in `.ASSUME DPP1:NOTHING`

  This indicates that nothing is assumed in the DPP register at that time. If a DPP register is assumed to contain nothing, the assembler does not implicitly use this DPP register for memory addressing. Also possible is: `.ASSUME NOTHING` This is the same as: `.ASSUME DPP1:NOTHING, DPP1:NOTHING,`

DPP2:NOTHING, DPP3:NOTHING This is the default which remains in effect until the first .ASSUME directive is found.

• SYSTEM keyword, as in .ASSUME DPP1:SYSTEM

This keyword enables the addressability of system ranges (via SFR) if a SFR is used in a virtual operand combination.

Search sequence

When you use a label that is assumed directly, via the section it is defined in or via the group of the section it is defined in, the following sequence is searched:

1. if the used label as a DPP assumed, this DPP is used

2. if the used label does not have a DPP assumed, but the section it is defined in does have a DPP assumed, the DPP on the section is used

3. if the used label does not have a DPP assumed and the section it is defined does not have a DPP assumed, the assume of a DPP on the group is used if present

Example

Specify an existing processor:

```
DESC1 .section far
AWORD .dw 0
DESC2 .section far
BYTE1 .db 0
DESC3 .section far
BYTE2 .db 0
CSEC  .section code
.ASSUME DPP0:DSEC1, DPP1:DSEC3
MOV   DPP0, #PAG DSEC1
MOV   DPP1, #PAG DSEC3
MOV   DPP2, #PAG DSEC2
.
.
MOV  R0, AWORD        ; The .assume covers the reference
.
.                     ; DPP0 points to the base of
.                     ; section DSEC1 that contains AWORD
.
MOV  RL1, DPP2:BYTE1  ; Explicit code. The page override
.                     ; operator covers the reference
MOV  RL1, BYTE1       ; Error!: No DPP register used and
.                     ; no ASSUME has been made
.
.
MOV  RL2, BYTE2       ; The .assume covers the reference
.
```

```
.                        ; DPP1 points to the base of
.                        ; section DSEC3 that contains BYTE2
```

When several DPPs are assumed to one *sectpart*, the lowest DPP number is used as DPP prefix. This also happens if, for example, both a label and the section it belongs to are assumed to different DPPs, or if both a section and the group it belongs to, are assumed to different DPPs:

```
      .ASSUME  DPP1:AGRP, DPP2:AVAR1
DSEC1 .section far, group( AGRP )
AVAR1 .dw 1
DESC2 .section far, group( AGRP )
    .
    .
    .
CSEC .section code
    .
    .
    .
    MOV R0, AVAR1  ; DPP1 is used for AVAR1
    .
    .
    .ASSUME DPP1:NOTHING
    MOV R0, AVAR1  ; DPP2 is used for AVAR1
    MOV R0, AGRP   ; DPP2 is used for AGRP
    .
    .
    RET
```

`.ASSUME` directives can forward reference a name. Also double forward references are allowed:

```
      .ASSUME  DPP0:DSEC1  ; Forward reference
      .ASSUME  DPP1:AVar   ; Double forward reference.
DSEC1 .section far
    .
    .
    .
AVar  .equ  wVar + 2
DSEC1 .section far
wVar  .dw 0
      .dw 0
```

.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
```

.CALLS

Syntax

.CALLS *'caller'*,*'callee'*

or

.CALLS *'caller'*,*''*,*ssk*,*usk*

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 usage count can be specified for the system stack (*ssk*) and the user stack (*usk*). The values specified are the stack usage in bytes at the time of the call including the return address.

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

This directive is generated by the C compiler. Normally you will not use it in hand-coded assembly.

Example

The function `_main` calls the function `_nfunc`:

```
.CALLS '_main', '_nfunc'
```

The function `_main()` uses 4 bytes on the system stack and no user stack:

```
.CALLS '_main','',4,0
```

.CGROUP, .DGROUP

Syntax

```
groupname  .CGROUP sectname [,sectname]...
groupname  .DGROUP sectname [,sectname]...
```

Description

With the .CGROUP directive you can create a group (*groupname*) of code sections. All sections within the same group will be placed within the same segment. With the .DGROUP directive you can create a group of data sections. All data sections with one group must be within the same space (section's space attribute). The group will be located as follows:

Space	Locate behavior
near	the whole group in the same page
far	the whole group in the same page
shuge	the whole group in the same segment
huge	no restrictions are made by the group, in LSL the sections can be selected with the group
bit	no restrictions are made by the group, in LSL the sections can be selected with the group

One special *sectname* in a data group is the SYSTEM section. When SYSTEM is grouped with the data group, the whole group will be placed in the SYSTEM page, page 3. The LSL file of the locator defines an empty SYSTEM section at the start of the system page to achieve this.

Example

```
CSEC1     .section   code
          .
          .
CSEC1     .ends

CSEC2     .section   code
          .
          .
CSEC2     .ends

CODEGRP   .CGROUP   CSEC1, CSEC2  ; Group combination of the CODE
                                  ; sections CSEC1 and CSEC2
```

.DBIT, .DB, .DW, .DL, .DLL

Syntax

```
[label] .DBIT argument[,argument]...
[label] .DB argument[,argument]...
[label] .DW argument[,argument]...
[label] .DL argument[,argument]...
[label] .DLL 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). For single bit initialization (.DBIT) the argument must be a positive absolute expression and each argument represents a bit to be initialized.

Multiple arguments are stored in successive byte locations. One or more arguments can be null (indicated by two adjacent commas), in which case the corresponding byte location will be filled with zeros.

The following table shows the number of bits initialized.

Directive	Bits	Alignment
.DBIT	1	1 bit
.DB	8	8 bit
.DW	16	16 bit
.DL	32	16 bit
.DLL	64	16 bit

The directive must be placed on an address that is aligned as listed in the table. A warning is issued if the directive is not aligned properly. You can use the .ALIGN directive to align the location counter.

When these directives are used in a BIT section, each argument initializes the number of bits defined for the used directive and the location counter of the current section is incremented with this number of bits.

The .DBIT directive can be used in a BIT section only. Each argument represents a bit to be initialized to 0 or 1. The location counter of the current section is incremented by a number of bits equal to the number of arguments.

The value of the arguments must be in range with the size of the directive; floating-point numbers are not allowed. If the evaluated argument is too large to be represented in a word / long / long long, 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:

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

Related Information

.DBFILL (Fill Block)

.DS (Define Storage)

.DBFILL, .DWFILL, .DLFILL, .DLLFILL

Syntax

```
[label]  .DBFILL count[,argument]
[label]  .DWFILL count[,argument]
[label]  .DLFILL count[,argument]
[label]  .DLLFILL count[,argument]
```

Description

With these directives the assembler allocates and initializes a block of memory filled with *argument*. The number of items in the block is defined by the constant expression *count*. The width of each item and the alignment of the block depends on the used directive:

Directive	Bits	Alignment
.DBFILL	8	8 bit
.DWFILL	16	16 bit
.DLFILL	32	16 bit
.DLLFILL	64	16 bit

The *argument* can be a single- or multiple-character string constant or an expression. If you omit the *argument*, the block is filled with zeros.

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

Example

```
DSEC .section far
.DB 84,101,115,116   ; initialize 4 bytes
.DBFILL 96,0xFF      ; reserve another 96 bytes, initialized with 0xFF
```

Related Information

.DB (Define Memory)

.DS (Define Storage)

.DBPTR, .DPPTR, .DSPTR

Syntax

```
[label]  .DBPTR argument[,argument]
[label]  .DPPTR argument[,argument]
[label]  .DSPTR argument[,argument]
```

Description

With these directives the assembler allocates and initializes pointer values in memory. These directives are included for backwards compatibility.

.DEBUG

Syntax

.DEBUG *section-name*[[,] **cluster** *name*]

Description

Create a DWARF debug section. Debug sections are not allocated by the linker. They contain high level language information generated by the compiler. This information is required for the debugger. The debug section names always start with a period as determined in the DWARF debug information specification for the C166 toolset. The sections contains constants and relocations referring to line numbers, register usage, variable lifetime and other debug information.

With '**cluster** *name*' this debug section is clustered with companion debug and code sections. It is used by the linker during removal of unreferenced sections. The name must be unique for this module (not for the application).

Normally you will not use this directive in hand-coded assembly.

Example

```
.DEBUG .debug_info
```

.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.

Example

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

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

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

```
.DS LEN
$MESSAGE(I,"The length is: LEN")
```

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

```
.DS 32
$MESSAGE(I,"The length is: 32")
```

Related Information

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

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

.DS, .DSBIT, .DSB, .DSW, .DSL, DSLL

Syntax

```
[label] .DS expression
[label] .DSBIT expression
[label] .DSB expression
[label] .DSW expression
[label] .DSL expression
[label] .DSLL expression
```

Description

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

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

The *expression* specifies the number of MAUs (Minimal Addressable Units) to be reserved, and how much the location counter will advance. The expression must evaluate to an integer greater than zero and cannot contain any forward references (symbols that have not yet been defined). In a bit section, the MAU size is 1, thus the `.DS` directive will initializes a number of bits equal to the result of the expression.

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

Directive	Reserved bits	Alignment
`.DSBIT`	1	1 bit
`.DSB`	8	8 bit
`.DSW`	16	16 bit
`.DSL`	32	16 bit
`.DSLL`	64	16 bit

The directive must be placed on an address that is aligned as listed in the table. A warning is issued if the directive is not aligned properly. You can use the `.ALIGN` directive to align the location counter.

Example

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

Related Information

`.DB` (Define Memory)

`.DBFILL` (Fill Block)

.END

Syntax

`.END`

Description

With the `.END` directive you tell the assembler that the end of the module is reached. The assembler will not process any lines following an `.END` directive. If the command line option **--require-end** is used the assembler will issue an error if the `.END` directive is not found before end of file. If a generator (e.g., a C compiler) stops generating before finishing the assembly file, the assembler can detect this by a missing `.END` directive.

The assembler does not allow a label with this directive.

Example

```
CSEC .section code
        ; source lines
     .END                   ; End of assembly module
```

Related Information

Assembler option **--require-end**

.EQU

Syntax

`symbol .EQU expression`

Description

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

The symbol defined with the `.EQU` gets a type depending on the resulting type of the expression. If the resulting type of the expression is none the symbol gets no type when the `.EQU` is used outside a section and it gets the type of the section when it is defined inside a section.

Example

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

`MYSYMBOL .EQU  0x4000`

Related Information

Section 3.7.4, *Symbol Types and Expression Types*

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

.EXTERN

Syntax

`.EXTERN` [`DPP`*x*`:`]*symbol* [`:`*type*]

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.

The *type* of the *symbol* is inherited from the section in which it is defined or from the directive used to define it. The assembler uses the type to check the symbol's use. In other words, if the symbol does not fit the instruction's operand, the assembler will issue a warning. If you do not specify the type information with the `.EXTERN` directive, the assembler will not check the use of the specified symbol.

You can use the DPP*x* prefix to specify the DPP register to be used to access the external symbol.

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 DPP2:AVAR:WORD ; extern declaration

CSEC .section code
    .
    .
    MOV R0, AVAR     ; AVAR is used here
    .
```

Related Information

See Section 3.7.4, *Symbol Types and Expression Types* for more information on the *type* keywords.

`.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.

The assembler does not allow a label with this directive. The type of the global symbol is determined by its definition.

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
```

.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 `AS166INC` 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. The assembler always opens an include file, even if the `.INCLUDE` directive is in between an inactive `.IF`/`.ENDIF`:

```
.if 0
.include "foo.asm"
.endif
```

This means that the include file always should be present.

Example

It is allowed to start a new section in an included file. If this file is included somewhere in another section, the contents of that section following the included file will belong to the section started in the include file:

```
; file incfile.asm

insect .section near
    .db 5
    .db 6
```

```
; file mainfile.asm

mainsect .section near
      .db 1
      .db 2
      .INCLUDE "incfile.asm"
      .db 3
      .db 4
```

The resulting sections have the following contents:

```
mainsect: 0x01 0x02
incsect:  0x05 0x06 0x03 0x04
```

.LABEL

Syntax

label **.LABEL** *type*

Description

Define a *label* of the specified *type*. The label is assigned the current location counter.

A label can be a code label, ending with a semicolon (e.g. `clab1:`), or a data label, without a semicolon.

Example

```
DSEC       .SECTION  NEAR
AWORD      .LABEL     WORD   ; label of type WORD
LOWBYTE    .DB  1
HBYTE      .LABEL     BYTE   ; label of type BYTE
HIGHBYTE   .DB  1
```

Related Information

See Section 3.7.4, *Symbol Types and Expression Types* for more information on the *type* keywords.

#line

Syntax

#[**line**] *linenumber* [*"filename"*]

Description

The line directive is the only directive not starting with a dot, but with a hash sign. It allows passing on line number information from higher level sources. This *linenumber* is used when generating errors. When this directive is encountered, the internal line number count is reset to the specified number and counting continues after the directive. The line after the directive is assumed to originate on the specified line number. The optional file name will, when specified, reset the module file name for purposes of error generation.

This directive is generated by the preprocessor phase of the C compiler. Normally you will not use it in hand-coded assembly.

Example

```
#line 1
```

.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:

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

The macro expands as follows:

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

Related Information

Section 3.10, *Macro Operations*

.DEFINE (Define a substitution string)

.MISRAC

Syntax

.MISRAC *string*

Description

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

Example

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

Related Information

Section 4.8.2, *C Code Checking: MISRA-C*

C compiler option **--misrac**

.PROC, .ENDP

Syntax

label **.PROC NEAR**
[[*label*] **.ENDP**]

label **.PROC FAR**
[[*label*] **.ENDP**]

label **.PROC INTNO** [[*name*]**=**][*number*]
[[*label*] **.ENDP**]

Description

Define a procedure with the name *label*. The following type of procedures can be defined:

Procedure type	Description
near	Near procedures are called using the CALLA instruction and must have a RETN as return instruction.
far	Far procedures are called using the CALLS instruction and must have a RETS as return instruction.
intno	Interrupt procedures, requiring RETI as return instruction. The interrupt can be assigned with a name and a number, used to define the interrupt vector table at link time.

The procedure type is applied to all labels that follow the .PROC directive until the procedure is ended. The *label* gets the defined procedure type. For interrupt functions the labels do not get a type because interrupt functions cannot be called.

The .ENDP ends the procedure, but is optional. The procedure also ends when a new .PROC is started in the same section or when the section ends.

Example

The following example defines and calls a far procedure:

```
GLOBALCODE .section code

AFARPROC   .PROC FAR        ; far procedure
           .
           .
           RETS             ; far return
AFARPROC   .ENDP

SPECSEC    .section code
           .
           .
           CALLS AFARPROC   ; far intra segment call.
```

Definition of an interrupt (trap) function:

```
_tfunc   .PROC INTNO tfunc_trap = -1
         .
         .
         RETI
```

.REGBANK

Syntax

```
bank-name .REGBANK [register-range]
```

Description

With the `.REGBANK` directive you can define a register bank with name *bank-name*. The registers used in the instructions must be defined in the `.REGBANK` directive. The assembler does **not** check this. The directive generates a section named *bank-name* with the iram section.

The label *bank-name* gets the type REGBANK and is placed at the location where R0 is positioned, even if R0 is not part of the register-range. The assembler checks if the GPRs being used in the source match those specified in the `.REGBANK` directive. Multiple `.REGBANK` directives per source file are allowed.

A section generated by the `.REGBANK` directive is defined from the lowest up to and including the highest register in the register range. If R0 is not in the register range, the section label will lie outside of the regbank section. When two modules use the same register bank name, the register banks are overlaid (section with MAX attribute). The linker overlays the start of the register banks, even if that location does not refer to the same register. This can be used for simple register bank sharing as follows:

```
module1:
bankname .REGBANK R0-R5

module2:
bankname .REGBANK R10-R15
```

In this case, the section `bankname` is overlaid. Both modules use a local label called `bankname` when they need to load the context pointer. The final register bank has a size of 6 words, pointing to either R0-R5 for `module1` or to R10-R15 for `module2`.

The assembler allows multiple definitions of the same register bank (with the same register range) in one module, which results in a single register bank:

```
module3:
bankname .REGBANK R0-R5
bankname .REGBANK R0-R15    ; OK
bankname .REGBANK R5-R10    ; error
```

Complex register bank definitions without .REGBANK

To make complex register bank definitions it is recommended not to use the `.REGBANK` directive. Instead you should create an iram section. All symbols in such a section must get the type `regbank`. For example:

```
banks    .section iram
                     ;bank1   bank2    bank3
bank1    .label regbank
         .dsw    1       ; 0
bank2    .label regbank
         .dsw    1       ; 1       0
         .dsw    1       ; 2       1
```

```
        .dsw   1       ; 3       2
        .dsw   1       ; 4       3
bank3   .label regbank
        .dsw   1       ; 5       4       0
        .dsw   15      ;6-15    6-15    1-15
banks   .ends
```

Example

```
.NEW
```

.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)

.SECTION, .ENDS

Syntax

```
name .SECTION type[,attribute...][,'classname']
    ....
```

`[[[name] .ENDS]`

Description

Use this directive to define section names and declaration attributes and for activating the section. For compatibility reasons, the commas between the operands of the .SECTION directive are optional. By default, the assembler tries to resume a previous section with the same name. If no such section exists, it creates a new section.

The *name* specifies the name of the section. The *type* operand specifies the section's space and must be one of:

Type	Description
BIT	Located in the bit addressable area. The section locator counts in bits.
BITA	Located in the bit addressable area. The section locator counts in bytes.
IRAM	Located in the internal RAM.
NEAR	Data section in a 64 kB address space. The underlying pages can be mapped anywhere in memory.
FAR	Data section that can be located anywhere in memory. Sections cannot be larger than 16 kB and cannot cross page boundaries.
SHUGE	Data section that can be located anywhere in memory. Sections cannot be larger than 64 kB and cannot cross segment boundaries.
HUGE	Data section that can be located anywhere in memory.
CODE	Code section that can be located anywhere in memory. Sections cannot be larger than 64 kB and cannot cross segment boundaries. The type of the labels in a code section depends on the used .PROC directive. Labels defined in a code section outside the .PROC directive get the type FAR. This can be overruled with the .LABEL directive and .PROC directive.

The section type and attributes are case insensitive.

The defined *attribute*s are:

Attribute	Description
AT *address*	Locate the section at the given *address*.
BYTE	Make the section byte aligned.
CLASS '*classname*'	Adds the *classname* to section name, separated with a dot (*name.classname*).

Attribute	Description
CLEAR	Sections are zeroed at startup.
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).
DWORD	Align the section on a double word boundary.
GLOBAL	Tells the linker to combine sections with the same name and attributes to one single section.
GROUP '*group*'	Used to group sections, for example for placing in the same page. You can also use the `.CGROUP` or `.DGROUP` directive for this.
INIT	Defines that the section contains initialization data, which is copied from ROM to RAM at program startup.
INPAGE	Defines that the section must be located within a page and cannot cross page boundaries. Only applicable to near, far, shuge and huge sections.
INSEGMENT	Defines that the section must be located with a segment and cannot cross page boundaries. Only applicable to shuge and huge sections.
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
NEW	Tells the assembler to start a new section. Use this for example when this section's name is equal to a previously started section with the same or different attributes.
NOCLEAR	Not zeroed at startup. This is a default attribute for data sections.
NOINIT	Defines that the section contains no initialization data. This is a default attribute for all data sections.
PAGE	Align the section on a page boundary. When you want to start locating at the first address in the page, you must also define the symbol `__PAGE_START=0` to the linker. You can do this in the LSL file with `#define __PAGE_START 0` or you can specify command line option **-D__PAGE_START=0** to the linker. See also the file `arch_c166.lsl` in the directory `include.lsl`.
PRIVATE	Tells the linker not to combine this section with sections with the same name and attributes. This is the default.
PROTECT	Tells the linker to exclude a section from unreferenced section removal and duplicate section removal.
ROMDATA	Section contains data to be placed in ROM
SEGMENT	Align the section on a segment boundary.
WORD	Make the section word aligned. This is the default for all sections.

Section names

The GROUP attribute results in an extended section name. This is similar to using the `.CGROUP` or `.DGROUP` directives. The *classname* is added to the section's name and makes it possible to select sections in the LSL file for locating. The name resulting from the section directive is as follows:

```
 section-name[.class-name][@group]
```

Example

```
DSEC   .SECTION near,init
TAB2   .DW 8    ; initialized section
DSEC   .ENDS

ABSSEC .SECTION far, at 0x100
     ; absolute section
```

Related Information

Section 3.7.4, *Symbol Types and Expression Types*.

.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)

.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"
```

Related Information

-

.UNDEF

Syntax

```
.UNDEF symbol
```

Description

With the `.UNDEF` directive you can undefine a macro or 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)

3.9.2. Assembler Controls

Controls start with a **$** as the first character on the line. Unknown controls are ignored after a warning is issued. The arguments of controls can optionally be enclosed in braces (). All controls have abbreviations of 2 characters (or 4 characters for the $no.. variant).

Overview of assembler controls

Control	Description
$[NO]ASMLINEINFO	Generate source line information for assembly files
$[NO]CHECK	Enable or disable the check for a silicon bug
$DATE	Set the date in the list file page header
$[NO]DEBUG	Control debug information generation
$EJECT	Generate form feed in list file page header
$[NO]LIST	Print / do not print source lines to list file
$[NO]LOCALS	Control generation of local symbols
$MESSAGE	Programmer generated message
$[NO]OPTIMIZE	Control optimization
$PAGELENGTH	Set list file page length
$PAGEWIDTH	Set list file page width
$[NO]PAGING	Control pagination of list file
$[NO]RETCHECK	Control checking of return instruction
$SAVE / $RESTORE	Save and restore the current value of the $LIST / $NOLIST controls
$[NO]SYMB	Control generation of symbolic debug information
$TABS	Specify tab size
$TITLE	Set program title in header of assembly list file
$[NO]WARNING	Enable or disable a warning

$ASMLINEINFO / $NOASMLINEINFO

Syntax

`$ASMLINEINFO`
`$NOASMLINEINFO`

Default

`$NOASMLINEINFO`

Abbreviation

`$AL / $NOAL`

Description

With the `$ASMLINEINFO` control the assembler generates assembly level debug information. This matches the effect of the **--debug-info=+asm** (**-ga**) command line option. When you use the command line option, it sets the default, but the control will override its effect.

Example

```
$ASMLINEINFO
    ;generate line and file debug information
    MOV R0, R12
$NOASMLINEINFO
    ;stop generating line and file information
```

Related Information

Assembler option **--debug-info**

Assembler control **$DEBUG**

$CHECK / $NOCHECK

Syntax

`$CHECK(`*number*`)`
`$NOCHECK`[`(`*number*`)`]

Default

`$NOCHECK` (for all numbers)

Abbreviation

`$CH / $NOCH`

Description

The `$CHECK` control enables the check for silicon problem with index *number*. For the list of numbers, see Chapter 18, *CPU Problem Bypasses and Checks*. You can use the `$NOCHECK` control to disable the check of a specific silicon problem number.

Example

To specify to check for silicon bug 18 from within the assembly source, specify:

```
$CHECK(18)
```

Related Information

Assembler option **--silicon-bug**

Chapter 18, *CPU Problem Bypasses and Checks*

$DATE

Syntax

`$DATE(`*string*`)`

Abbreviation

`$DA`

Description

This control sets the date as subtitle of the list file page header. When no `$DATE` is used the assembler uses the date and time when the list file was generated. The string argument of the `$DATE` control is not checked for a valid date, in fact any string can be used.

Example

```
; Feb 03 2006 in header of list file
$date('Feb 03 2006')
```

Related Information

Assembler option **--list-file**

$DEBUG / $NODEBUG

Syntax

`$DEBUG`
`$NODEBUG`

Default

`$NODEBUG`

Abbreviation

`$DB / $NODB`

Description

With the `$DEBUG` control you enable the assembler to generate debug information. If no high-level language debug information is present, debug information on assembly level is generated. This control also generates debug information on local symbols. This matches the effect of the **--debug-info=+local,+smart** (**-gls**) command line option. When you use the command line option, it sets the default, but the control will override its effect.

Example

```
$DEBUG
   ;generate smart debug information and information on local symbols
   MOV R0, R12
```

Related Information

Assembler option **--debug-info**

Assembler control **$ASMLINEINFO**

Assembler control **$LOCALS**

Assembler control **$SYMB**

$EJECT

Syntax

`$EJECT`

Default

A new page is started when the page length is reached.

Abbreviation

`$EJ`

Description

If you generate a list file with the assembler option **--list-file**, with the `$EJECT` control the list file generation advances to a new page by inserting a form feed. The new page is started with a new page header. The `$EJECT` control generates empty lines when `$NOPAGING` is set.

Example

```
    .       ; assembler source lines
    .
$EJECT      ; generate a formfeed
    .
```

Related Information

Assembler option **--list-file**

Assembler control **$PAGING**

$LIST / $NOLIST

Syntax

```
$LIST
$NOLIST
```

Default

```
$LIST
```

Abbreviation

```
$LI / $NOLI
```

Description

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

Example

```
    ... ; source line in list file
$NOLIST
    ... ; source line not in list file
$LIST
    ... ; source line also in list file
```

Related Information

Assembler option **--list-file**

Assembler control **$SAVE / $RESTORE**

$LOCAL / $NOLOCALS

Syntax

```
$LOCALS
$NOLOCALS
```

Default

```
$LOCALS
```

Abbreviation

```
$LC / $NOLC
```

Description

With the `$LOCALS` control the assembler generates debug information on local symbol records. This matches the effect of the **--debug-info=+local** (**-gl**) command line option. When you use the command line option, it sets the default, but the control will override its effect.

Example

```
$NOLOCALS  ; the assembler keeps no local symbol information
           ; of the following source lines
```

Related Information

Assembler option **--debug-info**

Assembler control **$ASMLINEINFO**

Assembler control **$DEBUG**

Assembler control **$SYMB**

$MESSAGE

Syntax

$MESSAGE(*type*, {*str*|*exp*}[,{*str*|*exp*}]... **)**

Abbreviation

$ME

Description

With the $MESSAGE control 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. Each subsequent argument is printed directly after the previous argument.

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

Example

```
$MESSAGE(I,'Generating tables')

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

    .DEFINE LONG "SHORT"
$MESSAGE(I,'This is a LONG string')
$MESSAGE(I,"This is a LONG string")
```

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

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

$OPTIMIZE / $NOOPTIMIZE

Syntax

```
$OPTIMIZE
$NOOPTIMIZE
```

Default

```
$OPTIMIZE
```

Abbreviation

```
$OP / $NOOP
```

Description

With these controls you can turn on or off conditional jump optimization, expansion of generic instructions and jump chain optimizations. This control overrules the **--optimize** (**-O**) command line option.

Please note that all instructions that have a word and a byte variant (and sometimes a bit variant) are implemented as generic instructions. Use the mnemonic ending in 'W' for word variants and the mnemonic ending in 'B' for byte variants. Combining $NOOPTIMIZE and generic instructions causes syntax errors.

Example

```
$noop
   ; turn optimization off
   ; source lines
$op
   ; turn optimization back on
   ; source lines
```

Related Information

Assembler option **--optimize**

$PAGELENGTH

Syntax

$PAGELENGTH(*pagelength*[,*blanktop*,*blankbtm*] **)**

Default

$PAGELENGTH(72,0,0)

Abbreviation

$PL

Description

If you generate a list file with the assembler option **--list-file**, the $PAGELENGTH control sets the number of lines in a page in the list file and the top and bottom margins of a page.

The arguments may be any positive absolute integer expression, and must be separated by commas.

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*) $\leq$ (*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*) $\leq$ (*pagelength* - 10).

Example

```
$PL(55)     ; page length is 55 with no top and bottom margin
$PL(55,4,2) ; page length is 55 with 4 blank lines at the top
            ; and 2 at the bottom
```

Related Information

Assembler option **--list-file**

Assembler control **$PAGEWIDTH**

$PAGEWIDTH

Syntax

$PAGEWIDTH(*pagewidth*[,*blankleft*] **)**

Default

$PAGEWIDTH(132,0)

Abbreviation

$PW

Description

If you generate a list file with the assembler option **--list-file**, the $PAGEWIDTH control sets the width of a page in the list file and the left margin of the page.

The arguments may be any positive absolute integer expression, and must be separated by a comma.

pagewidth	Number of columns per line. The default is 132, the minimum is 40.
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*.

Example

`$PW(80,8)   ; set the pagewidth to 80 characters and start with 8 spaces`

Related Information

Assembler option **--list-file**

Assembler control **$PAGELENGTH**

$PAGING / $NOPAGING

Syntax

```
$PAGING
$NOPAGING
```

Default

```
$NOPAGING
```

Abbreviation

```
$PA / $NOPA
```

Description

If you generate a list file with the assembler option **--list-file**, you can use these controls to turn the generation of form feeds in the list file on or off.

Example

```
$pa
; turn paging on: formfeed before each page header
```

Related Information

Assembler option **--list-file**

Assembler control **$EJECT**

$RETCHECK / $NORETCHECK

Syntax

```
$RETCHECK
$NORETCHECK
```

Default

```
$NORETCHECK
```

Abbreviation

```
$RC / $NORC
```

Description

$RETCHECK turns on the checking for the correct return instruction from a routine. For example, an interrupt function must be returned from with a RETI instruction. If the assembler finds another return instruction within the interrupt function an error will be generated. $NORETCHECK turns off the checking for the correct return instruction from a subroutine.

Example

```
$RETCHECK

PRC .PROC INTNO isr=1
    ; source lines
    RETS  ; error, RETI expected
```

The assembler will give an error on the RETS instruction, because an interrupt procedure must be ended with a RETI instruction.

Related Information

Assembler option **--retcheck**

$SAVE / $RESTORE

Syntax

```
$SAVE
$RESTORE
```

Abbreviation

```
$SA / $RE
```

Description

The $SAVE control stores the current value of the $LIST / $NOLIST controls onto a stack. The $RESTORE control restores the most recently saved value; it takes effect starting at the next line. You can nest $SAVE controls to a depth of 16.

Example

```
$nolist
  ; source lines
$save             ; save values of $LIST / $NOLIST

$list

$restore          ; restore value ($nolist)
```

Related Information

Assembler option **--list-file**

Assembler control **$LIST**

$SYMB / $NOSYMB

Syntax

```
$SYMB
$NOSYMB
```

Default

```
$NOSYMB
```

Abbreviation

```
$SB / $NOSB
```

Description

With the $SYMB control the assembler enables generation of high-level language debug information. This matches the effect of the **--debug-info=+hll** (**-gh**) command line option. When you use the command line option, it sets the default, but the control will override its effect.

Example

```
$SYMB
    ;generate high-level language debug information
```

Related Information

Assembler option **--debug-info**

Assembler control **$DEBUG**

$TABS

Syntax

$TABS(_number_**)**

Default

$TABS(8)

Abbreviation

$TA

Description

$TABS specifies the tab positions in the list file. For each tab character a maximum of _number_ of blanks is inserted until the next tab position is reached.

Example

```
$TABS(4)
; use 4 spaces for a tab
```

Related Information

Assembler option **--list-file**

$TITLE

Syntax

$TITLE([*string*]**)**

Default

The module name.

Abbreviation

$TT

Description

The $TITLE initializes the program title to the *string* specified in the operand field. The program title will be printed after the banner at the top of all succeeding pages of the source listing until another $TITLE control is encountered. An exception to this is the first $TITLE control, which sets the title of the first and following pages in the listing until the next $TITLE control is encountered.

A $TITLE with no string argument causes the current title to be blank. The title is initially the name of the module. The $TITLE control will not be printed in the source listing.

Example

```
$TITLE("This is the new title in the list file")
```

Related Information

Assembler option **--list-file**

$WARNING / $NOWARNING

Syntax

```
$WARNING( number )
NOWARNING( number )
```

Default

```
$WARNING
```

Abbreviation

```
$WA / $NOWA
```

Description

This control allows you to enable or disable all or individual warnings. The *number* argument can have the following values:

0	Select no warning messages
1, **2**	Select all warning messages
>2	Select a specific warning message number.

Example

```
$NOWARNING(1)          ; disable all warnings
$WARNING(1)            ; enable all warnings
$NOWARNING(735)        ; disable warning W 735
```

Related Information

Assembler option **--no-warnings**

3.10. Macro Operations

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

Some patterns contain variable entries which change for each repetition of the pattern. Others are subject to conditional assembly.

When a macro is called, the assembler executes the macro and replaces the call by the resulting in-line source statements. 'In-line' means that all replacements act as if they are on the same line as the macro call. The generated statements may contain substitutable arguments. The statements produced by a macro can be any processor instruction, almost any assembler directive, or any previously-defined macro. Source statements resulting from a macro call are subject to the same conditions and restrictions as any other statements.

Macros can be nested. The assembler processes nested macros when the outer macro is expanded.

3.10.1. Defining a Macro

The first step in using a macro is to define it.

The definition of a macro consists of three parts:

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

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

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

A macro definition takes the following form:

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

For more information on the definition see the description of the `.MACRO` directive.

3.10.2. Calling a Macro

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

```
[label] macro_name [argument[,argument]...]  [; comment]
```

where,

label	An optional label that corresponds to the value of the location counter at the start of the macro expansion.
macro_name	The name of the macro. This may not start in the first column.

argument	One or more optional, substitutable arguments. Multiple arguments must be separated by commas.
comment	An optional comment.

The following applies to macro arguments:

- Each argument must correspond one-to-one with the formal arguments of the macro definition. If the macro call does not contain the same number of arguments as the macro definition, the assembler issues a warning.

- If an argument has an embedded comma or space, you must surround the argument by single quotes (').

- You can declare a macro call argument as null in three ways:

  - enter delimiting commas in succession with no intervening spaces

    ```
    macroname ARG1,,ARG3 ; the second argument is a null argument
    ```

  - terminate the argument list with a comma, the arguments that normally would follow, are now considered null

    ```
    macroname ARG1,      ; the second and all following arguments are null
    ```

  - declare the argument as a null string

- No character is substituted in the generated statements that reference a null argument.

3.10.3. Using Operators for Macro Arguments

The assembler recognizes certain text operators within macro definitions which allow text substitution of arguments during macro expansion. You can use these operators for text concatenation, numeric conversion, and string handling.

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

Example: Argument Concatenation Operator - \

Consider the following macro definition:

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

The macro is called as follows:

```
MAC_A 0,1
```

The macro expands as follows:

```
   mov r0,#1
```

The macro preprocessor substitutes the character '0' 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 character 'r'.

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

```
   mov 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 0,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
   mov 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:  mov   r0,^addr
      .ENDM
```

The macro is called as follows:

```
LOCAL:
        INIT LOCAL
```

The macro expands as:

```
LOCAL__M_L000001: mov  r0,LOCAL
```

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

```
LOCAL__M_L000001: mov   r0,LOCAL__M_L000001
```

3.11. Generic Instructions

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

The assembler knows the following generic instructions:

CALL

* CALLR -> If the target address operand has the type NEAR and the address fits within the relative range.

* CALLA -> If the target address operand has the type NEAR and the address does not fit within the relative range.

* CALLS -> If the target address operand type is FAR or if 2 non-address operands are used (segment and segment offset).

* CALLI -> If an indirect operand is used.

* PCALL ->If the first operand is a register to be pushed.

If a condition code is omitted, the cc_UC condition code is used.

JMP

* JMPR -> If the target address fits within the relative range within the same section or when the target address is a label with the SHORT type.

- JMPA -> If the target address has the type NEAR or if the target address operand does not fit within the relative range.

- JMPS -> If the target address operand has the type FAR or if 2 non-address operands are used (segment and segment offset).

- JMPI -> If the operand is indirect.

If a condition code is specified only JMPR or JMPA can be chosen and FAR target address operands are not allowed. If a condition code is omitted, the `cc_UC` condition code is used.

JB

Results in JB if the target address is within the relative range. If the target is not within the relative range, a combination of JNB/JMPA (NEAR type operand) or JNB/JMPS (FAR type operand) is used.

JNB

Results in JNB if the target address is within the relative range. If the target is not within the relative range, a combination of JB/JMPA (NEAR type operand) or JB/JMPS (FAR type operand) is used.

RET

Results in a return instruction, depending on the procedure type specified with the `.PROC` directive:

- RETN -> For `.proc near`

- RETS -> For `.proc far`

- RETI -> For `.proc intno`

Jump optimizations that cannot be done by the assembler are postponed to the linker.

RETV

RETV is a virtual return instruction. It disables generation of the warning message "procedure *procedure-name* contains no RETurn instruction". No code is generated for this instruction. You can put this instruction just before the `.ENDP` directive of the procedure that caused the warning message.

ADD, ADDC, AND, CMP, CPL, MOV, NEG, OR, SUB, SUBC, XOR

When word, byte or (for some) bit operands are supplied, these instructions result in their respective word, byte or bit variants. Forcing a specific variant is done by appending a '**W**' for word-variant or a '**B**' for byte-variant or by prepending a '**B**' for the bit-variant. This yields four variants of each instruction.

Example with the AND:

- AND -> Generic, can result in ANDW, ANDB or BAND depending on its operands.

- ANDW -> Word instruction, requires word operands.

- ANDB -> Byte instruction, requires byte operands.

- BAND -> Bit instruction, requires bit operands.

When both word and byte variants are possible, the word variant is chosen. This occurs for the double indirect addressing modes (i.e. `mov [R1], [R2]`) and the REG, IMM addressing mode (i.e. `mov DPP0,#2`). If word aligned labels are used, the word variant is chosen, even though the byte variant would fit as well (i.e. `mov DPP0, _label`).

Chapter 4. Using the C Compiler

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

TASKING VX-toolset for C166 under Eclipse can use the internal builder (default) or the TASKING makefile generator (external builder) to build your entire embedded project, from C source till the final ELF/DWARF object file which serves as input for the debugger.

Although in Eclipse you cannot run the C compiler separately from the other tools, this section discusses the options that you can specify for the C compiler.

On the command line it is possible to call the C compiler separately from the other tools. However, it is recommended to use the control program for command line invocations of the toolset (see Section 9.1, *Control Program*). With the control program it is possible to call the entire toolset with only one command line.

The C compiler takes the following files for input and output:
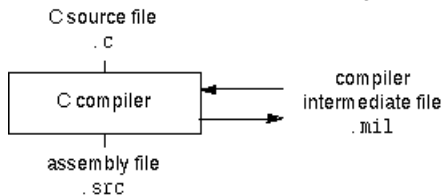
This chapter first describes the compilation process which consists of a *frontend* and a *backend* part. Next it is described how to call the C compiler and how to use its options. An extensive list of all options and their descriptions is included in Section 12.2, *C Compiler Options*. Finally, a few important basic tasks are described, such as including the C startup code and performing various optimizations.

4.1. Compilation Process

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

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

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

Frontend phases

1. The preprocessor phase:

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

   Performs target processor independent and dependent optimizations which operate on the Low level Intermediate Language.

4. The code generation/formatter phase:

   This phase reads through the LIL operations to generate assembly language output.

4.2. Calling the C Compiler

C166 under Eclipse can use the internal builder (default) or the TASKING makefile generator (external builder) to build your entire project. After you have built your project, the output files are available in a subdirectory of your project directory, depending on the active configuration you have set in the **C/C++ Build » Settings** page of the **Project » Properties for** dialog.

Building a project under Eclipse

You have several ways of building your project:

- Build Selected File(s) (⊞). This compiles and assembles the selected file(s) without calling the linker.

  1. In the C/C++ Projects view, select the files you want to compile.

  2. Right-click in the C/C++ Projects view and select **Build Selected File(s)**.

- Build Individual Project ().

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

- Rebuild Project (). This builds every file in the project whether or not a file has been modified since the last build. A rebuild is a clean followed by a build.

    1. Select **Project » Clean...**

    2. Enable the option **Start a build immediately** and click **OK**.

- Build Automatically. This performs a build of all projects whenever any project file is saved, such as your makefile.

    This way of building is not recommended for C/C++ development, but to enable this feature select **Project » Build Automatically** and ensure there is a check mark beside the **Build Automatically** menu item. In order for this option to work, you must also enable option **Build on resource save (Auto build)** on the **Behaviour** tab of the **C/C++ Build** page of the **Project » Properties for** dialog.

See also Section 4.7, *Influencing the Build Time*.

Select a target processor (core)

Processor options affect the invocation of all tools in the toolset. In Eclipse you only need to set them once. Based on the target processor, the compiler includes a special function register file. This is a regular include file which enables you to use virtual registers that are located in memory.

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

    The Properties dialog appears.

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

    In the right pane the Processor page appears.

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

To access the C/C++ compiler options

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

    The Properties dialog appears.

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

    In the right pane the Settings appear.

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

4. Select the sub-entries and set the options in the various pages.

> Note that the C/C++ compiler options are used to create an object file from a C or C++ file. The options you enter in the Assembler page are not only used for hand-coded assembly files, but also for intermediate assembly files.

You can find a detailed description of all C compiler options in Section 12.2, *C Compiler Options*.

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

`c166` [[*option*]... [*file*]...]...

4.3. The C Startup Code

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

- *Initialization code.* This code is executed when the program is initiated and before the function `main()` is called. It initializes the processor's registers and the application C variables.

- *Exit code.* This controls the close down of the application after the program's main function terminates.

To add the C startup code to your project

When you create a new project with the New C/C++ Project wizard (**File » New » TASKING VX-toolset for C166 C/C++ Project**), fill in the dialogs and enable the option **Add C startup code to the project** in the following dialog (this is the default setting).
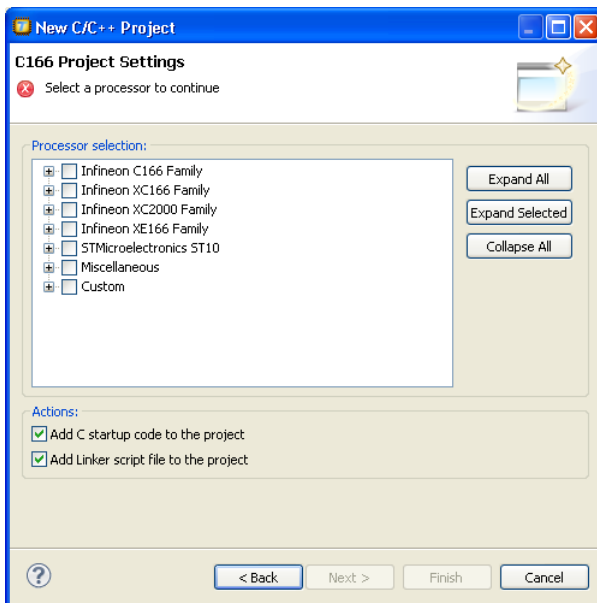
This adds the files `cstart.c` and `cstart.h` to your project. These files are copies of `lib/src/cstart.c` and `include/cstart.h`. If you do not add the startup code here, you can always add it later with **File » New » cstart.c/cstart.h Files**.

To change the C startup code configuration and registers

The project Properties dialog contains two pages where you can change the C startup code.

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

   The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Startup Configuration** or **Startup Registers**.

   In the right pane the Startup Configuration page or Startup Registers page appears.
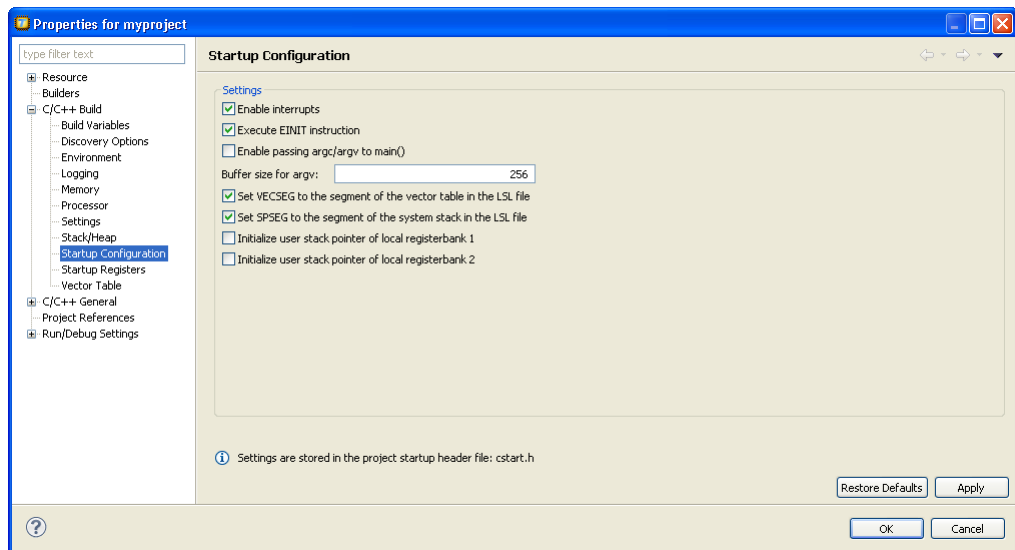
3. On the **Startup Configuration** page, you can make changes to the C startup code configuration. For example, you can choose to disable interrupts.

4. On the **Startup Registers** page, you can specify the registers and their settings that must be known to the startup code. Enable the option **Initialize in startup code** to add a register setting to the startup code. For some registers you can also enable the option **Use mask for initialization**. A define ending on `_DO_MASK` is set in the file `cstart.h`. This will mask out bits in the register in `cstart.c` that could have been set by hardware. If you made changes to a register and you want to reset the register to its original value, click on the **Default** button. See Section 4.3.1, *ICACHE Support* for an example.

5. Click **OK**.

   The file cstart.h in your project is updated with the new values.

The values of the startup registers for a project are only set to their default values at project creation for the at that time selected processor.

When you switch to a different processor afterwards, in the **Project » Properties for » C/C++ Build » Processor** property page, the registers are not set to their defaults again. The reason for that is that you may have set specific values in the startup registers that you want to keep.
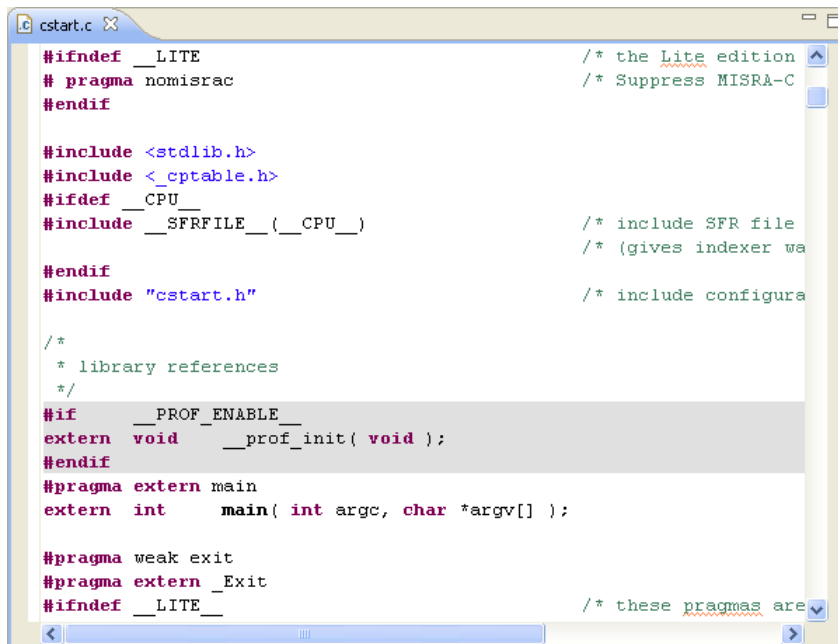
If you want to set all registers to their default values for the selected processor, you can do that any time by clicking on the **Restore Defaults** button on the **Project » Properties for » C/C++ Build » Startup Registers** property page.

When you use Import Board Configuration wizard to import (register) settings required for a certain board, only the registers needed to get the board going in the default situation are changed.

To change the C startup code in Eclipse manually

1. Double-click on the file `cstart.c` or `cstart.h`.

    The file opens in the editor area.

2. You can edit the C startup code directly in the editor.

    *A * appears in front of the name of the file to indicate that the file has changes.*

3. Click 💾 or select **File » Save** to save the changes.

4.3.1. ICACHE Support

Some of the newer Infineon devices, such as the XC2268I have a cache on the flash memory, the so-called ICACHE. To use the ICACHE it must be enabled in the C startup code and the linker script file must be modified to get sections located in the memory ranges that are covered by the ICACHE (see Section 8.7.11, *ICACHE Support and Named Memory Mappings*).

To use the ICACHE it must be enabled on the processor at startup. You can do this by setting the `ICACHE_CTRL.ICEN` bit. The C startup code, `cstart.c`, can initialize the `ICACHE_CTRL` SFR.

To enable the ICACHE in Eclipse

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

    The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Startup Registers**.

    In the right pane the Startup Registers page appears.
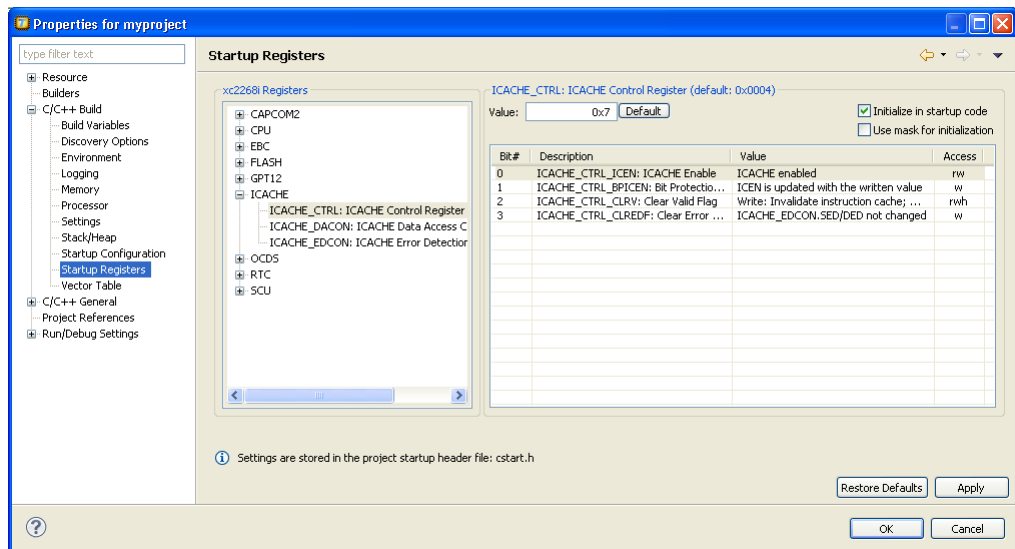
3. Expand the ICACHE registers.

4. Select `ICACHE_CTRL`.

5. Set the field **ICACHE_CTRL_IEN: ICACHE_Enable** to "`ICACHE enabled`"

6. Set the field **ICACHE_CTRL_BPICEN: Bit Protection for ICACHE_Enable** to "`ICEN is updated with the written value`"

7. Enable option **Initialize in startup code** to add the setting to the startup code.

8. Click **OK**.

   The file cstart.h in your project is updated with the new values.

> The `ICACHE_CTRL` register is only available on devices that have an ICACHE. If you do not see this register, then select the appropriate processor first.

To enable the ICACHE manually

1. Make sure the latest `cstart.c` and `cstart.h` files are included in your project.

2. Open the file `cstart.h` in an editor.

3. Search for `__ICACHE_CTRL`.

4. Change the macro definitions as follows:

```
#define __ICACHE_CTRL_INIT     1
#define __ICACHE_CTRL_VALUE    0x7
```

When the `__ICACHE_CTRL_INIT` macro is set to 1 the `ICACHE_CTRL` SFR will be initialized at startup The `__ICACHE_CTRL_VALUE` macro defines the initialization value.

> If you already have a `cstart.c` and `cstart.h` file from a version v2.4 or older in your project, you must update these files to obtain ICACHE support. You can find the files in the product's installation directory: `include/cstart.h` and `lib/src/cstart.c`.

4.4. How the Compiler Searches Include Files

When you use include files (with the `#include` statement), you can specify their location in several ways. The compiler searches the specified locations in the following order:

1. If the `#include` statement contains an absolute pathname, the compiler looks for this file. If no path or a relative path is specified, the compiler looks in the same directory as the source file. This is only possible for include files that are enclosed in "".

> This first step is not done for include files enclosed in <>.

2. When the compiler did not find the include file, it looks in the directories that are specified in the **C/C++ Compiler » Include Paths** page in the **C/C++ Build » Settings » Tool Settings** tab of the Project Properties dialog (equivalent to the **-I** command line option).

3. When the compiler did not find the include file (because it is not in the specified include directory or because no directory is specified), it looks in the path(s) specified in the environment variable `C166INC`.

4. When the compiler still did not find the include file, it finally tries the default include directory relative to the installation directory.

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:

```
c166 -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 `C166INC` 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 `C166INC` and then in the default `include` directory.

4.5. Compiling for Debugging

Compiling your files is the first step to get your application ready to run on a target. However, during development of your application you first may want to debug your application.

To create an object file that can be used for debugging, you must instruct the compiler to include symbolic debug information in the source file.

To include symbolic debug information

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

    The Properties dialog appears.

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

    In the right pane the Settings appear.

3. On the Tool Settings tab, select **C/C++ Compiler » Debugging**.

4. Select **Default** in the **Generate symbolic debug information** box.

Debug and optimizations

Due to different compiler optimizations, it might be possible that certain debug information is optimized away. Therefore, if you encounter strange behavior during debugging it might be necessary to reduce the optimization level, so that the source code is still suitable for debugging. For more information on optimization see Section 4.6, *Compiler Optimizations*.

Invocation syntax on the command line (Windows Command Prompt)

The invocation syntax on the command line is:

```
c166 -g file.c
```

4.6. Compiler Optimizations

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

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

    The Properties dialog appears.

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

    In the right pane the Settings appear.

3. On the Tool Settings tab, select **C/C++ Compiler » Optimization**.

4. Select an optimization level in the **Optimization level** box.

    or:

    In the **Optimization level** box select **Custom optimization** and enable the optimizations you want on the Custom optimization page.

Optimization levels

The TASKING C compiler offers four optimization levels and a custom level, at each level a specific set of optimizations is enabled.

*   **Level 0 - No optimization**: No optimizations are performed. The compiler tries to achieve a 1-to-1 resemblance between source code and produced code. Expressions are evaluated in the order written in the source code, associative and commutative properties are not used.

*   **Level 1 - Optimize**: Enables optimizations that do not affect the debug-ability of the source code. Use this level when you encounter problems during debugging your source code with optimization level 2.

*   **Level 2 - Optimize more (default)**: Enables more optimizations to reduce the memory footprint and/or execution time. This is the default optimization level.

*   **Level 3 - Optimize most**: This is the highest optimization level. Use this level when your program/hardware has become too slow to meet your real-time requirements.

*   **Custom optimization**: you can enable/disable specific optimizations on the Custom optimization page.

Optimization pragmas

If you specify a certain optimization, all code in the module is subject to that optimization. Within the C source file you can overrule the C compiler options for optimizations with `#pragma optimize` *flag* and `#pragma endoptimize`. Nesting is allowed:

```
#pragma optimize e     /* Enable expression
...                        simplification               */
... C source ...
```

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

The compiler optimizes the code between the pragma pair as specified.

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

4.6.1. Generic Optimizations (frontend)

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

Branch prediction (option -O-predict/-O+predict)

A prediction is done if branches are likely to be taken or not. Based on this, other optimizations can take place.

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.
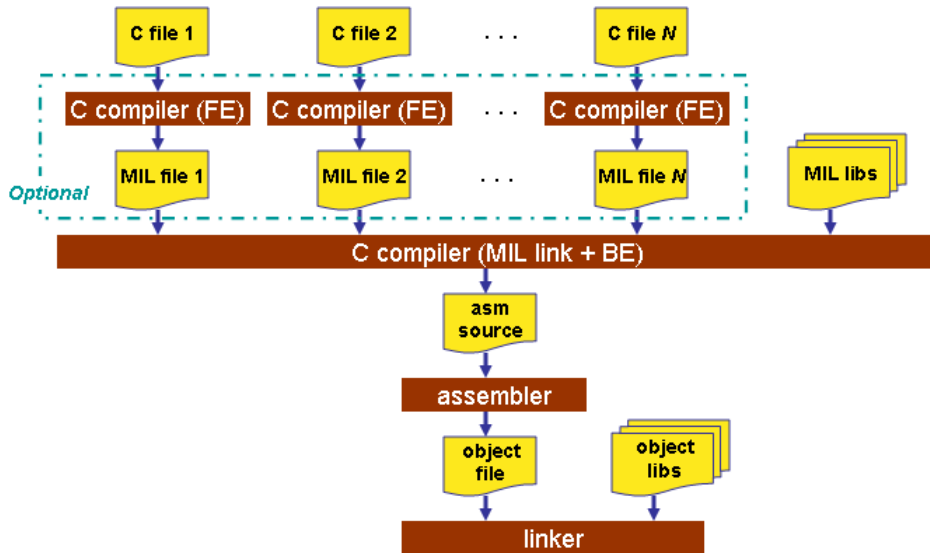
MIL splitting (option --mil-split)

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.
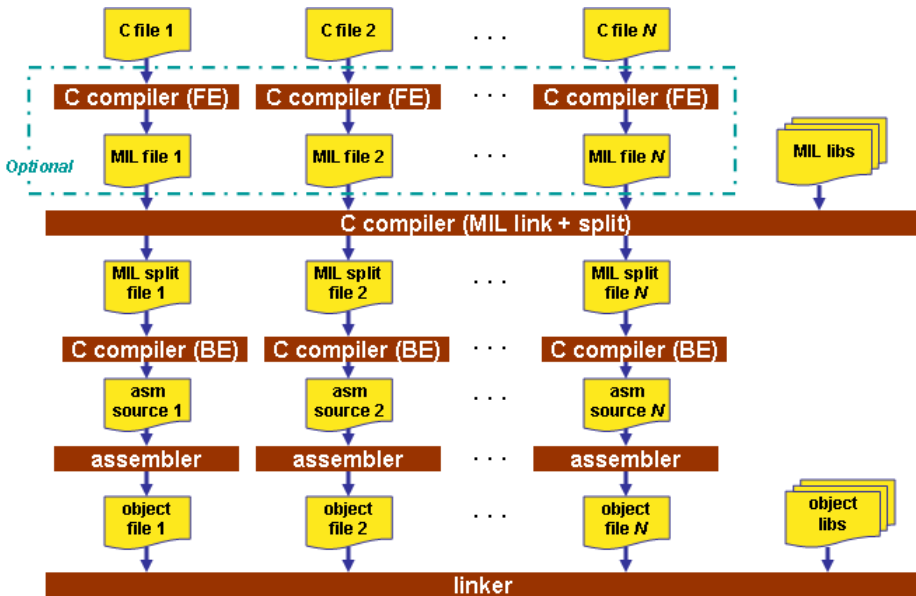
To read more about how MIL linking influences the build process of your application, see Section 4.7, *Influencing the Build Time*.

4.6.2. Core Specific Optimizations (backend)

Coalescer (option -Oa/-OA)

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

Interprocedural register optimization (option -Ob/-OB)

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

Peephole optimizations (option -Oy/-OY)

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

Instruction Scheduler (option -Ok/-OK)

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

First the instruction stream is partitioned into basic blocks. A new basic block starts at a label, or right after a jump instruction. Unschedulable instructions and, when **-Av** is enabled, instructions that access volatile objects, each get their own basic block. Next, the scheduler searches the instructions within a

basic block, looking for places where the pipeline stalls. After identifying these places it tries to rebuild the basic block using the existing instructions, while avoiding the pipeline stalls. In this process data dependencies between instructions are honoured.

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

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

Compaction is the opposite of inlining functions: large chunks of code that occur more than once, are transformed into a function. This reduces code size at the cost of execution speed.

Generic assembly optimizations (option -Og/-OG)

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

Automatic near data allocation (application wide) (option --automatic-near)

In the far, shuge and huge memory models this optimization tries to move objects to the near memory space automatically. In addition, pointers to these objects will also be qualified `near` automatically. Because near memory can be accessed more efficiently than far/shuge/huge this will save code and the generated code will be faster. This optimization can only be used together with the MIL linking or MIL splitting build process, because it needs application scope. Only objects and pointers that are in the default memory space are affected, objects and pointers explicitly qualified as `__far`/`__shuge`/`__huge` are not a candidate for this optimization.

Because the C compiler must allocate objects in the near memory space, it needs to know how much near memory is available, which parts of it are ROM and RAM, etc. To obtain this information, the C compiler reads the LSL file. This must be the same LSL file as the linker uses. The C compiler only considers the near memory space and expects it to be free. This will be verified using the LSL file. It is possible to define heap, stack, vector table and reserved areas, select them and locate them at an absolute address. Because the C compiler does not have information about assembly sections, it is not possible to select other sections, and try to locate them in the address range of the near memory space.

Limitations

For the qualification of pointers and objects this optimization uses a special pointer qualifier that is not available at C level: `__near32`. A `__near32` pointer behaves like a `__near` pointer, but will take 32 bits of storage in memory/stack. The upper 16 bits of a `__near32` pointer are not used. The reason for this storage inefficiency is that the `sizeof()` operator must return the same value for the pointer rewritten to 'near' by the automatic near data optimization as for a far/shuge/huge pointer.

When an object/pointer is rewritten by the optimization, the debugger will show a `__near32` pointer.

Other limitations of the automatic near data optimization are:

• pointers that are passed to a function in a variable argument list cannot be rewritten.

• pointers that are a struct/union member cannot be rewritten.

• the optimization cannot trace pointers with more than one indirection level.

Note that also pointers related to the pointers in the cases above will not be rewritten. For example (`test.c`):

```
struct
{
    char * p;
} s;
char * q;
char   c;

void main( void )
{
    q   = &c;
    s.p = q;
}
```

Invocation:

```
cc166 test.c -Mf --mil-split --automatic-near
```

Because '`q`' is assigned to '`s.p`' both pointers will be not be rewritten, and therefore '`c`' cannot be relocated to the `__near32` space. Because '`s`' and '`q`' are both unrelated, the storage of these objects will be moved to `__near32`.

4.6.3. Optimize for Size or Speed

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

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

Optimization hint: Optimizing for size has a speed penalty and vice versa. It takes an average of 42% more code to gain 8% speed (measured with the near model for xc16x, using option **-O2**). This is largely caused by the Code Compaction optimization. The advice is to optimize for size by default and only optimize those areas for speed that are critical for the application with respect to speed. Using the tradeoff options **-t0**, **-t1** and **-t2** globally for the application is not recommended.

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

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

    The Properties dialog appears.

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

In the right pane the Settings appear.

3. On the Tool Settings tab, select **C/C++ Compiler » Optimization**.

4. Select a trade-off level in the **Trade-off between speed and size** box.

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

Instruction Selection

Trade-off levels 0, 1 and 2: the compiler selects the instructions with the smallest number of cycles.

Trade-off levels 3 and 4: the compiler selects the instructions with the smallest number of bytes.

Switch Jump Chain versus Jump Table

Instruction selection for the switch statements follows different trade-off rules. A switch statement can result in a jump chain or a jump table. The compiler makes the decision between those by measuring and weighing bytes and cycles. This weigh is controlled with the trade-off values:

Trade-off value	Time	Size
0	100%	0%
1	75%	25%
2	50%	50%
3	25%	75%
4	0%	100%

Subscript Strength Reduction

The trade-off limits the total number of additional pointers of a particular type in a particular loop.

The C166 has 14 registers that you can use as 16-bit pointers (14 word registers) or as 32-bit pointers (7 double-word registers).

The performance always increases when more subscript pointers can be allocated for an ideal situation. Ideal is when no registers are needed for other objects than subscripts. This is rarely the case, therefore we control the number of word registers with the trade-off option.

Trade-off value	Number of word registers
0	12
1	10
2	8
3	6
4	4

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:

```
        jmp    _2          ;; unconditional jump to loop test at bottom
_3:
        subw   _a,ONES
        addw   r3,#0x1
_2:                        ;; loop entry point
        cmpw   r3,r2
        jmp    cc_slt,_3
```

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

```
        movw   r11,_a
        subw   r2,r3
        cmpw   r2,#0x0     ;; test for at least one loop iteration
        jmp    cc_sle,_4   ;; can be omitted when number of iterations is known
```

```
_3:                             ;; loop entry point
        addw   r11,#0x1
        subw   r2,#0x1
        jmp    cc_ne,_3
_4:
```

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	360	22
1	270	19
2	180	16
3	90	13
4	0	10

For example with trade-off value 1, the compiler inlines all functions that are smaller or equal to 19 internal compiler units. After that the compiler tries to inline even more functions as long as the function will not grow more than 270%.

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.

MAC Optimizations

The compiler tries to judge what the gain will be if MAC instructions are used instead of regular instructions. This is measured in bytes and cycles. For the resulting gain, the size in bytes and cycles are weighed with the trade-off setting:

Trade-off value	Time	Size
0	100%	0%
1	75%	25%
2	50%	50%
3	25%	75%
4	0%	100%

The estimated execution frequency of an instruction is multiplied by the number of cycles.

When the compiler generates MAC instructions, it has the following favors:

- Trade-off levels 0, 1 and 2: speed

- Trade-off levels 3 and 4: size

Code Compaction

Trade-off levels 0 and 1: code compaction is disabled.

Trade-off level 2: only code compaction of matches outside loops.

Trade-off level 3: code compaction of matches outside loops, and matches inside loops of patterns that have an estimate execution frequency lower or equal to 10.

Trade-off level 4: code compaction of matches outside loops, and matches inside loops of patterns that have an estimate execution frequency lower or equal to 100.

For loops where the iteration count is unknown an iteration count of 10 is assumed.

For the execution frequency the compiler also accounts nested loops.

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

4.7. 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.

SFR File

SFR files for recent devices like XC2287M define such a large number of SFRs that compiling the SFR file alone already takes up a significant part of the build time. There are two ways to reduce the build time:

- Disable the automatic inclusion of the SFR file and include the SFR file only in the source modules where the SFRs are used, with a `#include` directive. You can disable the automatic inclusion of the SFR file with option **--no-tasking-sfr** of the tools. In Eclipse you can find this option on the "**C/C++ Compiler » Preprocessing**" and the "**Assembler » Preprocessing**" pages.

  When you include the SFR file in the source, be aware that the SFR files are in the `sfr` subdirectory of the include files, so you must use: `#include <sfr/regxc2287m.sfr>`

- Use the alternative SFR file format. The product's `include/sfr` directory also contains SFR files with the suffix `.asfr`. These alternative SFR files do not include a macro definition for each bit-field and SFRs must be accessed using the correct struct/union fields, for example:

  ```
  PSW.U = 0x0010;   (instead of PSW = 0x0010;)
  PSW.B.N = 0;      (instead of N = 0;)
  ```

  You can select the alternative SFR files with the C compiler option **--alternative-sfr-file**. In Eclipse you can find this option on the **C/C++ Compiler » Preprocessing** page.

Of course you can combine both ways: disable automatic SFR file inclusion and use `#include` `<sfr/regxc2287m.asfr>` where SFRs are used.

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 in Eclipse are:

- **Build for application wide optimizations (MIL linking)**

  This enables MIL linking. The build process changes: the C files are translated to intermediate code (MIL files) and the generated MIL files of the whole project are linked together by the C compiler. The next step depends on the setting of the option below.

- **Application wide optimization mode: Optimize more/Build slower**

  When this option is enabled, the compiler runs the code generator immediately on the completely linked MIL stream, which represents the entire application. This way the code generator can perform several optimizations, such as "code compaction", at application scope. But this also requires significantly more memory and requires more time to generate code. Besides that, it is no longer possible to do incremental builds. With each build the full MIL linking phase and code generation has to be done, even with the smallest change that would in a normal build (not MIL linking) require only a single module to be translated.

- **Application wide optimization mode: Optimize less/Build faster**

  When this option is disabled, the compiler splits the MIL stream after MIL linking in separate modules. This allows the code generation to be performed for the modified modules only, and will therefore be faster than with the other option enabled. Although the MIL stream is split in separate modules after MIL linking, it still may happen that modifying a single C source file results in multiple MIL files to be compiled. This is a natural result of global optimizations, where the code generated for multiple modules was affected by the change.

In general, if you do not need code compaction, for example because you are optimizing fully for speed, it is recommended to choose **Optimize less/Build faster**.

Application wide automatic near allocation

The C compiler option **--automatic-near**, in Eclipse enabled on page **C/C++ Compiler » Allocation** with option **Application wide automatic near data allocation**, requires MIL linking. As described in the previous section, MIL linking will already increase the build time. The automatic near allocation itself also increases the build time.

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**, which is used by Eclipse, has a feature to build jobs in parallel. This means that multiple modules can be compiled in parallel. With today's multi-core processors this means that each core can be fully utilized. In practice even on single core machines the compile time decreases when using parallel jobs. On multi-core machines the build time even improves further when specifying more parallel jobs than the number of cores.

In Eclipse you can control the parallel build behavior:

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

    The Properties dialog appears.

2. In the left pane, select **C/C++ Build**.

    In the right pane the C/C++ Build page appears.

3. On the Behaviour tab, select **Use parallel build**.

4. You can specify the number of parallel jobs, or you can use an optimal number of jobs. In the last case, **amk** will fork as many jobs in parallel as cores are available.

Number of Sections

The linker speed depends on the number of sections in the object files. The more sections, the longer the locating will take. You can decrease the link time by creating output sections in the LSL file. For example:

```
section_layout ::code
{
  group (ordered)
  {
    section "code_output1" ( size = 64k, attributes = x, fill=0xFF,
                             overflow = "code_output2")
    {
      select "*__cocofun*";
    }
  }
}
```

4.8. 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.

4.8.1. 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 19, *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 19, *CERT C Secure Coding Standard*.

To apply CERT C code checking to your application

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

    The Properties dialog appears.

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

    In the right pane the Settings appear.

3. On the Tool Settings tab, select **C/C++ Compiler » CERT C Secure Coding**.

4. Make a selection from the **CERT C secure code checking** list.

5. If you selected **Custom**, expand the **Custom CERT C** entry and enable one or more individual recommendations/rules.

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

```
c166 --cert={all | name [-name],...]
```

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

4.8.2. 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 20, *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

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

   The Properties dialog appears.

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

   In the right pane the Settings appear.

3. On the Tool Settings tab, select **C/C++ Compiler » MISRA-C**.

4. Select the **MISRA-C version** (2004 or 1998).

5. In the **MISRA-C checking** box select a MISRA-C configuration. Select a predefined configuration for conformance with the required rules in the MISRA-C guidelines.

6. (Optional) In the **Custom 2004** or **Custom 1998** entry, specify the individual rules.

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

```
c166 --misrac={all | number [-number],...]
```

4.9. C Compiler Error Messages

The C compiler reports the following types of error messages in the Problems view of Eclipse.

F (Fatal errors)

After a fatal error the compiler immediately aborts compilation.

E (Errors)

Errors are reported, but the compiler continues compilation. No output files are produced unless you have set the C compiler option **--keep-output-files** (the resulting output file may be incomplete).

W (Warnings)

Warning messages do not result into an erroneous assembly output file. They are meant to draw your attention to assumptions of the compiler for a situation which may not be correct. You can control warnings in the **C/C++ Build » Settings » Tool Settings » C/C++ Compiler » Diagnostics** page of the **Project » Properties for** menu (C compiler option **--no-warnings**).

I (Information)

Information messages are always preceded by an error message. Information messages give extra information about the error.

S (System errors)

System errors occur when internal consistency checks fail and should never occur. When you still receive the system error message

```
S9##: internal consistency check failed - please report
```

please report the error number and as many details as possible about the context in which the error occurred.

Display detailed information on diagnostics

1. From the **Window** menu, select **Show View » Other » General » Problems**.

    The Problems view is added to the current perspective.

2. In the Problems view right-click on a message.

    A popup menu appears.

3. Select **Detailed Diagnostics Info**.

    A dialog box appears with additional information.

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

```
c166 --diag=[format:]{all | number,...]
```

Chapter 5. Using the C++ Compiler

This chapter describes the compilation process and explains how to call the C++ compiler. You should be familiar with the C++ language and with the ISO C language.

The C++ compiler can be seen as a preprocessor or front end which accepts C++ source files or sources using C++ language features. The output generated by the C++ compiler (**cp166**) is intermediate C, which can be translated with the C compiler (**c166**).

The C++ compiler is part of a complete toolset, the TASKING VX-toolset for C166. For details about the C compiler see Chapter 4, *Using the C Compiler*.

The C++ compiler takes the following files for input and output:
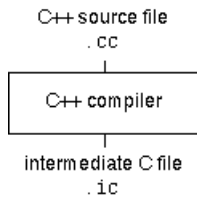
Although in Eclipse you cannot run the C++ compiler separately from the other tools, this section discusses the options that you can specify for the C++ compiler.

On the command line it is possible to call the C++ compiler separately from the other tools. However, it is recommended to use the control program for command line invocations of the toolset (see Section 9.1, *Control Program*). With the control program it is possible to call the entire toolset with only one command line. Eclipse also uses the control program to call the C++ compiler. Files with the extensions `.cc`, `.cpp` or `.cxx` are seen as C++ source files and passed to the C++ compiler.

The C++ compiler accepts the C++ language of the ISO/IEC 14882:1998 C++ standard, with some minor exceptions documented in Chapter 2, *C++ Language*. It also accepts embedded C++ language extensions.

The C++ compiler does no optimization. Its goal is to produce quickly a complete and clean parsed form of the source program, and to diagnose errors. It does complete error checking, produces clear error messages (including the position of the error within the source line), and avoids cascading of errors. It also tries to avoid seeming overly finicky to a knowledgeable C or C++ programmer.

5.1. Calling the C++ Compiler

Under Eclipse you cannot run the C++ compiler separately. However, you can set options specific for the C++ compiler. After you have built your project, the output files are available in a subdirectory of your project directory, depending on the active configuration you have set in the **C/C++ Build » Settings** page of the **Project » Properties for** dialog.

Building a project under Eclipse

You have several ways of building your project:

- Build Selected File(s) ( ). This compiles and assembles the selected file(s) without calling the linker.

1. In the C/C++ Projects view, select the files you want to compile.

2. Right-click in the C/C++ Projects view and select **Build Selected File(s)**.

- Build Individual Project ( ).

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

- Rebuild Project ( ). This builds every file in the project whether or not a file has been modified since the last build. A rebuild is a clean followed by a build.

  1. Select **Project » Clean...**

  2. Enable the option **Start a build immediately** and click **OK**.

- Build Automatically. This performs a build of all projects whenever any project file is saved, such as your makefile.

  This way of building is not recommended for C/C++ development, but to enable this feature select **Project » Build Automatically** and ensure there is a check mark beside the **Build Automatically** menu item.

Select a target processor (core)

Processor options affect the invocation of all tools in the toolset. In Eclipse you only need to set them once. Based on the target processor, the compiler includes a special function register file. This is a regular include file which enables you to use virtual registers that are located in memory.

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

   The Properties dialog appears.

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

   In the right pane the Processor page appears.

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

To access the C/C++ compiler options

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

   The Properties dialog appears.

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

   In the right pane the Settings appear.

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

4. Select the sub-entries and set the options in the various pages.

Note that C++ compiler options are only enabled if you have added a C++ file to your project, a file with the extension `.cc`, `.cpp` or `.cxx`.

Note that the options you enter in the Assembler page are also used for intermediate assembly files.

You can find a detailed description of all C++ compiler options in Section 12.3, *C++ Compiler Options*.

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

cp166 [[*option*]... [*file*]...]...

5.2. How the C++ Compiler Searches Include Files

When you use include files (with the `#include` statement), you can specify their location in several ways. The C++ compiler searches the specified locations in the following order:

1. If the `#include` statement contains an absolute pathname, the C++ compiler looks for this file. If no path or a relative path is specified, the C++ compiler looks in the same directory as the source file. This is only possible for include files that are enclosed in "".

   This first step is not done for include files enclosed in <>.

2. When the C++ compiler did not find the include file, it looks in the directories that are specified in the **C/C++ Compiler » Include Paths** page in the **C/C++ Build » Settings » Tool Settings** tab of the Project Properties dialog (equivalent to the **--include-directory** (**-I**) command line option).

3. When the C++ compiler did not find the include file (because it is not in the specified include directory or because no directory is specified), it looks in the path(s) specified in the environment variable `CP166INC`.

4. When the C++ compiler still did not find the include file, it finally tries the default `include.cpp` and `include` directory relative to the installation directory.

5. If the include file is still not found, the directories specified in the **--sys-include** option are searched.

If the include directory is specified as **-**, e.g., **-I-**, the option indicates the point in the list of **-I** or **--include-directory** options at which the search for file names enclosed in < ... > should begin. That is, the search for < ... > names should only consider directories named in **-I** or **--include-directory** options following the **-I-**, and the directories of items 3 and 4 above. **-I-** also removes the directory containing the current input file (item 1 above) from the search path for file names enclosed in " ... ".

An include directory specified with the **--sys-include** option is considered a "system" include directory. Warnings are suppressed when processing files found in system include directories.

If the filename has no suffix it will be searched for by appending each of a set of include file suffixes. When searching in a given directory all of the suffixes are tried in that directory before moving on to the

next search directory. The default set of suffixes is, no extension and `.stdh`. The default can be overridden using the **--incl-suffixes** command line option. A null file suffix cannot be used unless it is present in the suffix list (that is, the C++ compiler will always attempt to add a suffix from the suffix list when the filename has no suffix).

Example

Suppose that the C++ source file `test.cc` contains the following lines:

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

You can call the C++ compiler as follows:

```
cp166 -Imyinclude test.cc
```

First the C++ compiler looks for the file `stdio.h` in the directory `myinclude` relative to the current directory. If it was not found, the C++ compiler searches in the environment variable `CP166INC` and then in the default `include` directory.

The C++ compiler now looks for the file `myinc.h`, in the directory where `test.cc` is located. If the file is not there the C++ compiler searches in the directory `myinclude`. If it was still not found, the C++ compiler searches in the environment variable `CP166INC` and then in the default `include.cpp` and `include` directories.

5.3. C++ Compiler Error Messages

The C++ compiler reports the following types of error messages in the Problems view of Eclipse.

F (Fatal errors)

Catastrophic errors, also called 'fatal errors', indicate problems of such severity that the compilation cannot continue. For example: command-line errors, internal errors, and missing include files. If multiple source files are being compiled, any source files after the current one will not be compiled.

E (Errors)

Errors indicate violations of the syntax or semantic rules of the C++ language. Compilation continues, but object code is not generated.

W (Warnings)

Warnings indicate something valid but questionable. Compilation continues and object code is generated (if no errors are detected). You can control warnings in the **C/C++ Build » Settings » Tool Settings » C/C++ Compiler » Diagnostics** page of the **Project » Properties for** menu (C++ compiler option **--no-warnings**).

R (Remarks)

Remarks indicate something that is valid and probably intended, but which a careful programmer may want to check. These diagnostics are not issued by default. Compilation continues and object code is

generated (if no errors are detected). To enable remarks, enable the option **Issue remarks on C++ code** in the **C/C++ Build » Settings » Tool Settings » C/C++ Compiler » Diagnostics** page of the **Project » Properties for** menu (C++ compiler option **--remarks**).

S (Internal errors)

Internal compiler errors are caused by failed internal consistency checks and should never occur. However, if such a 'SYSTEM' error appears, please report the occurrence to Altium. Please include a small C++ program causing the error.

Message format

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

```
cp166 E0020: ["test.cc" 3] identifier "name" is undefined
```

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

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

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

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

```
cp166 E0308: ["test.cc" 4] more than one instance of overloaded
         function "f" matches the argument list:
            function "f(int)"
            function "f(float)"
            argument types are: (double)
```

In some cases, some additional context information is provided; specifically, such context information is useful when the C++ compiler issues a diagnostic while doing a template instantiation or while generating a constructor, destructor, or assignment operator function. For example:

```
cp166 E0265: ["test.cc" 7] "A::A()" is inaccessible
         detected during implicit generation of "B::B()" at line 7
```

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

Termination Messages

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

```
n errors detected in the compilation of "file".

1 catastrophic error detected in the compilation of "file".

n errors and 1 catastrophic error detected in the compilation of "file".
```

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

```
Error limit reached.
```

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

```
Compilation terminated.
```

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

```
Compilation aborted
```

is written at the end of a compilation that was prematurely terminated because of an internal error. Such an error indicates an internal problem in the compiler. If such an internal error appears, please report the occurrence to Altium. Please include a small C++ program causing the error.

Chapter 6. Profiling

Profiling is the process of collecting statistical data about a running application. With these data you can analyze which functions are called, how often they are called and what their execution time is. This chapter describes the TASKING profiling method with code instrumentation techniques and static profiling.

6.1. What is Profiling?

Profiling is a collection of methods to gather data about your application which helps you to identify code fragments where execution consumes the greatest amount of time.

TASKING supplies a number of profiler tools each dedicated to solve a particular type of performance tuning problem. Performance problems can be solved by:

- Identifying time-consuming algorithms and rewrite the code using a more time-efficient algorithm.

- Identifying time-consuming functions and select the appropriate compiler optimizations for these functions (for example, enable loop unrolling or function inlining).

- Identifying time consuming loops and add the appropriate pragmas to enable the compiler to further optimize these loops.

A profiler helps you to find and identify the time consuming constructs and provides you this way with valuable information to optimize your application.

TASKING employs various schemes for collecting profiling data, depending on the capabilities of the target system and different information needs.

6.1.1. Methods of Profiling

There are several methods of profiling: recording by an instruction set simulator, profiling with code instrumentation techniques (dynamic profiling) and profiling by the C compiler at compile time (static profiling). Each method has its advantages and disadvantages.

Profiling by an instruction set simulator

One way to gather profiling information is built into the instruction set simulator (ISS). The ISS records the time consumed by each instruction that is executed. The debugger then retrieves this information and correlates the time spent for individual instructions to C source statements.

Advantages

- it gives (cycle) accurate information with extreme fine granularity

- the executed code is identical to the non-profiled code

Disadvantages

- the method requires an ISS as execution environment

Profiling using code instrumentation techniques (Dynamic Profiling)

The TASKING C compiler has an option to add code to your application which takes care of the profiling process. This is called code instrumentation. The gathered profiling data is first stored in the target's memory and will be written to a file when the application finishes execution or when the function `__prof_cleanup()` is called.

Advantages

• it can give a complete call graph of the application annotated with the time spent in each function and basic block

• this profiling method is execution environment independent

• the application is profiled while it executes on its aimed target taking real-life input

Disadvantage

• instrumentation code creates a significant run-time overhead, and instrumentation code and gathered data take up target memory

This method provides a valuable complement to the other two methods and is described into more detail below.

Profiling estimation by the C compiler (Static Profiling)

The TASKING C compiler has an option to generate static profile information through various heuristics and estimates. The profiling data produced this way at compile time is stored in an XML file, which can be processed and displayed using the same tools used for dynamic (run-time) profiling.

Advantages

• it can give a give a quick estimation of the time spent in each function and basic block

• this profiling method is execution environment independent

• the application is profiled at compile time

• it requires no extra code instrumentation, so no extra run-time overhead

Disadvantage

• it is an estimation by the compiler and therefore less accurate than dynamic profiling

This method also is described into more detail below.

6.2. Profiling using Code Instrumentation (Dynamic Profiling)

Profiling can be used to determine which parts of a program take most of the execution time.

Once the collected data are presented, it may reveal which pieces of your code execute slower than expected and which functions contribute most to the overall execution time of a program. It gives you also information about which functions are called more or less often than expected. This information not

only reveal design flaws or bugs that had otherwise been unnoticed, it also reveals parts of the program which can be effectively optimized.

Important considerations

The dynamic profiling method adds code to your original application which is needed to gather the profiling data. Therefore, the code size of your application increases. Furthermore, during the profiling process, the gathered data is initially stored into dynamically allocated memory of the target. The heap of your application should be large enough to store this data. Since code instrumentation is done by the compiler, assembly functions used in your program do not show up in the profile.

The profiling information is collected during the actual execution of the program. Therefore, the input of the program influences the results. If a part/function of the program is not activated while the program is profiled, no profile data is generated for that part/function.

When you use dynamic profiling on a target board, function `clock()` must be added to the application to get profiling timer results. You can add the file `timer.c` from the example Profiling project to your project as a starting point.

It is *not* possible to profile applications that are compiled with the optimization code compaction (C compiler option **--optimize=+compact**). Therefore, when you turn profiling on, the compiler automatically disables parts of the code compaction optimization.

Overview of steps to perform

To obtain a profile using code instrumentation, perform the following steps:

1. Compile and link your program with profiling enabled

2. Execute the program to generate the profile data

3. Display the profile

First you need a completed project. If you are not using your own project, use the `profiling` example as described below.

1.   From the **File** menu, select **Import...**

    The Import dialog appears.

2.   Select **TASKING C/C++ » TASKING C166 Example Projects** and click **Next**.

3.   In the **Example projects** box, disable all projects except `profiling` .

4.   Click **Finish**.

    The `profiling` project should now be visible in the C/C++ view.

6.2.1. Step 1: Build your Application for Profiling

The first step is to add the code that takes care of the profiling, to your application. This is done with C compiler options:

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

     The Properties for profiling dialog box appears.

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

     In the right pane the Settings appear.

3.   On the Tool Settings tab, expand the **C/C++ Compiler** entry and select **Debugging**.

4.   Enable one or more of the following **Generate profiling information** options (the sample `profiling` project already has profiling options enabled).

     •   **for block counters** (not in combination with Call graph or Function timers)

     •   **to build a call graph** (not in combination with Block counters)

     •   **for function counters**

     •   **for function timers** (not in combination with Block counters/Function counters)

     > Note that the more detailed information you request, the larger the overhead in terms of execution time, code size and heap space needed. The option Generate symbolic debug information (**--debug**) does not affect profiling, execution time or code size.

     Block counters (not in combination with Call graph or Function timers)

     This will instrument the code to perform basic block counting. As the program runs, it will count how many time it executed each branch of each if statement, each iteration of a for loop, and so on. Note that though you can combine Block counters with Function counters, this has no effect because Function counters is only a subset of Block counters.

     Call graph (not in combination with Block counters)

     This will instrument the code to reconstruct the run-time call graph. As the program runs it associates the caller with the gathered profiling data.

     Function counters

     This will instrument the code to perform function call counting. This is a subset of the basic Block counters.

     Function timers (not in combination with Block counters/Function counters)

     This will instrument the code to measure the time spent in a function. This includes the time spent in all called functions (callees).

     For the command line, see the C compiler option **--profile** (**-p**).

     Profiling is only possible with optimization levels 0, 1 and 2. So:

5.   Open the **Optimization** page and set the **Optimization level** to **2 - Optimize more**.

6. Click **OK** to apply the new option settings and rebuild the project ().

6.2.1.1. Profiling Modules and C Libraries

Profiling individual modules

It is possible to profile individual C modules. In this case only limited profiling data is gathered for the functions in the modules compiled without the profiling option. When you use the suboption **Call graph**, the profiling data reveals which profiled functions are called by non-profiled functions. The profiling data does not show how often and from where the non-profiled functions themselves are called. Though this does not affect the flat profile, it might reduce the usefulness of the call graph.

Profiling C library functions

Eclipse and/or the control program will link your program with the standard version of the C library. Functions from this library which are used in your application, will not be profiled. If you do want to incorporate the library functions in the profile, you must set the appropriate C compiler options in the C library makefiles and rebuild the library.

6.2.1.2. Linking Profiling Libraries

When building your application, the application must be linked against the corresponding profile library. Eclipse (or the control program) automatically select the correct library based on the profiling options you specified. However, if you compile, assemble and link your application manually, make sure you specify the correct library.

See Section 8.3, *Linking with Libraries* for an overview of the (profiling) libraries.

6.2.2. Step 2: Execute the Application

Once you have compiled and linked the application for profiling, it must be executed to generate the profiling data. Run the program as usual: the program should run normally taking the same input as usual and producing the same output as usual. The application will run somewhat slower than normal because of the extra time spent on collecting the profiling data.

Eclipse has already made a default simulator debug configuration for your project. Follow the steps below to run the application on the TASKING simulator, using the debugger. (In fact, you can run the application also on a target board.)

1. From the **Debug** menu, select **Debug Configurations...**

   The Debug Configurations dialog appears.

2. Select the simulator debug configuration (**TASKING C/C++ Debugger » profiling.simulator**).

3. Click the **Debug** button to start the debugger and launch the profiling application.

   Eclipse will open the TASKING Debug perspective (as specified in the configuration) and asks for confirmation.

4. Click **Yes** to open the TASKING Debug perspective.

The TASKING Debug perspective opens while the application has stopped before it enters main()

5. In the Debug view, click on the ▮▶ (Resume) button.

    A file system simulation (FSS) view appears in which the application outputs the results.

    When the program has finished, the collected profiling data is saved (for details see 'After execution' below).

Startup code

The startup code initializes the profiling functions by calling the function `__prof_init()`. Eclipse will automatically make the required modifications to the startup code. Or, when you use the control program, this extracts the correct startup code from the C library.

If you use your own startup code, you must manually insert a call to the function `__prof_init` just before the call to `_main` and its stack setup.

An application can have multiple entry points, such as `main()` and other functions that are called by interrupt. This does not affect the profiling process.

Small heap problem

When the program does not run as usual, this is typically caused by a shortage of heap space. In this case a message is issued (when running with file system simulation, it is displayed on the Debug console). To solve this problem, increase the size of the heap. Information about the heap is stored in the linker script file (.lsl) file which is automatically added when a project is created.

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

    The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Stack/Heap**.

    In the right pane the Stack/Heap property page appears.

3. Enter larger values for **nheap** (near heap) and **hheap** (huge heap) and click **OK**.

    The project LSL file is updated automatically according to the new settings.

Presumable incorrect call graph

The call graph is based on the *compiled* source code. Due to compiler optimizations the call graph may therefor seem incorrect at first sight. For example, the compiler can replace a function call immediately followed by a return instruction by a jump to the callee, thereby merging the callee function with the caller function. In this case the time spent in the callee function is not recorded separately anymore, but added to the time spent in the caller function (which, as said before, now holds the callee function). This represents exactly the structure of your source in assembly but may differ from the structure in the initial C source.

After execution

When the program has finished (returning from `main()`), the exit code calls the function `__prof_cleanup(void)`. This function writes the gathered profiling data to a file on the host system using the debugger's file system simulation features. If your program does *not* return from `main()`, you can force this by inserting a call to the function `__prof_cleanup()` in your application source code. Please note the double underscores when calling from C code!

The resulting profiling data file is named `amon.prf`.

> If your program does not run under control of the debugger and therefore cannot use the file system simulation (FSS) functionality to write a file to the host system, you must implement a way to pass the profiling data gathered on the target to the host. Adapt the function `__prof_cleanup()` in the profiling libraries or the underlying I/O functions for this purpose.

6.2.3. Step 3: Displaying Profiling Results

After the function `__prof_cleanup()` has been executed, the result of the profiler can be displayed in the TASKING Profiler perspective. The profiling data in the file `amon.prf` is then converted to an XML file. This file is read and its information is displayed. To view the profiling information, open the TASKING Profiler perspective:

1. From the **Window** menu, select **Open Perspective** » **Other...**

   The Select Perspective dialog appears.

2. Select the **TASKING Profiler** perspective and click **OK**.

   The TASKING Profiler perspective opens.
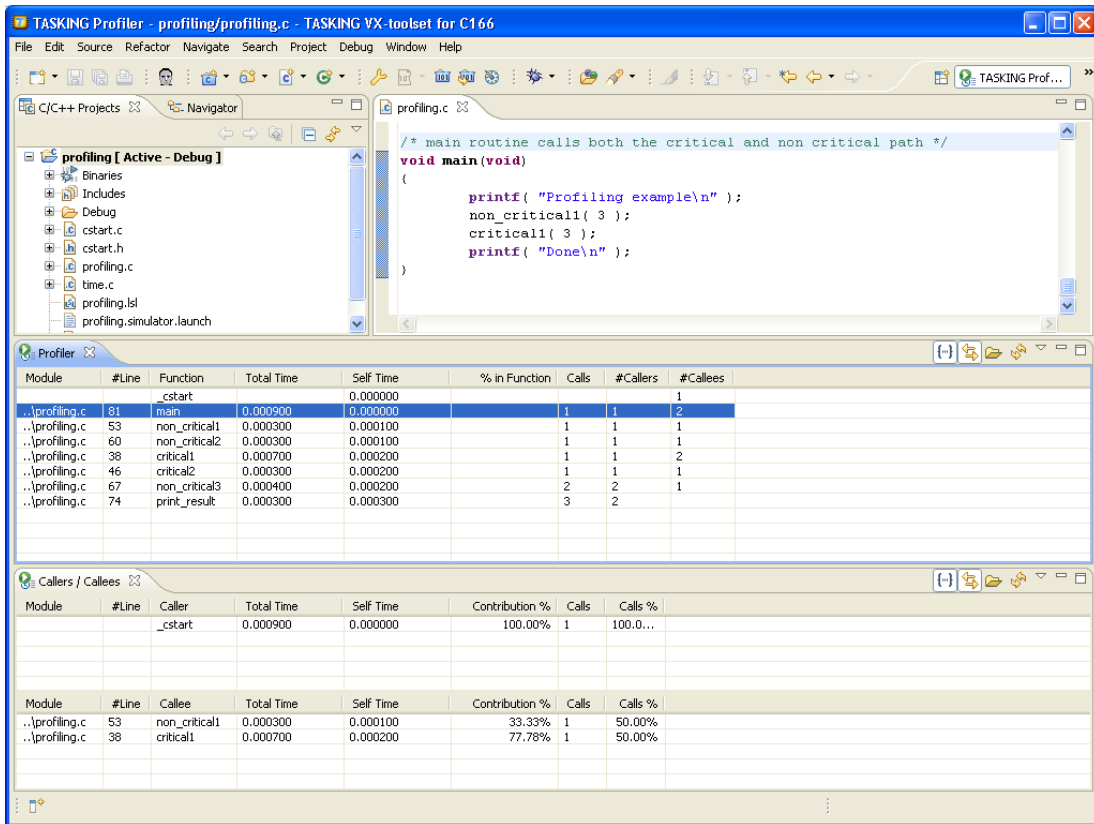
The TASKING Profiler perspective

The TASKING Profiler perspective contains the following Views:

Profiler view	Shows the profiling information of all functions in all C source modules belonging to your application.
Callers / Callees view	The first table in this view, the *callers* table, shows the functions that called the focus function.
	The second table in this view, the *callees* table, shows the functions that are called by the focus function.

• Clicking on a function (or on its table row) makes it the focus function.

• Double-clicking on a function, opens the appropriate C source module in the Editor view at the location of the function definition.

• To sort the rows in the table, click on one of the column headers.

The profiling information

Based on the profiling options you have set before compiling your application, some profiling data may be present and some may be not. The columns in the tables represent the following information:

Module	The C source module in which the function resides.
#Line	The line number of the function definition in the C source module.
Function	The function for which profiling data is gathered and (if present) the code blocks in each function. To show or hide the block counts, in the Profiler view click the **Menu** button (▽) and select **Show Block Counts**.
Total Time	The total amount of time in seconds that was spent in this function and all of its sub-functions.
Self Time	The amount of time in seconds that was spent in the function itself. This excludes the time spent in the sub-functions. So, self time = function's total time - total times of the called functions.
% in Function	This is the relative amount of time spent in this function, calculated as a percentage of the total application time. These should add up to 100%. The total application time is determined by taking the total time of the call graph. This is usually main or cstart. Example:

```
Total time of main: 0.002000
Self time of function foo:  0.000100
%in Function = (0.000100 / 0.002000) * 100 =  5%
```

Calls	Number of times the function has been executed.
#Callers	Number of functions by which the function was called.
#Callees	Number of functions that was actually called from this function.
Contribution %	In the caller table: shows for which part (in percent) the caller contributes to the time spent in the focus function. In the callee table: shows how much time the focus function has spent relatively in each of its callees.
Calls %	In the caller table: shows how often each callee was called as a percentage of all calls from the focus function. In the callee table: shows how often the focus function was called from a particular caller as a percentage of all calls to the focus function.

Common toolbar icons

Icon	Action	Description
{॥}	Show/Hide Block Counts	Toggle. If enabled, shows profiling information for block counters.
⇆	Link with Editor	Toggle. If enabled, updates the profiling information according to the active source file.
📂	Select Profiling File(s)	Opens a dialog where you can specify profiling files for display.

Icon	Action	Description
	Refresh Profiler Data	Updates the views with the latest profiling information.

Viewing previously recorded profiling results, combining results

Each time you run your application, new profiling information is gathered and stored in the file `amon.prf`. You can store previous results by renaming the file `amon.prf` (keep the extension `.prf`); this prevents the existing `amon.prf` from being overwritten by the new `amon.prf`. At any time, you can reload these profiling results in the profiler. You can even load multiple `.prf` files into the Profiler to view the combined results.

First, open the TASKING Profiler perspective if it is not open anymore:

1. In the Profiler view, click on the 📂 (Select Profiling File(s)) button.

   The Select Profiling File(s) dialog appears.

2. In the **Profiling Type** group box, select **Dynamic Profiling**.

3. In the **Profiling Files** group box, disable the option **Use default**.

4. Click the **Add...** button, select the `.prf` files you want to load and click **Open** to confirm your choice.

5. Make sure the correct symbol file is selected, in this example `profiling.elf`.

6. Click **OK** to finish.

6.3. Profiling at Compile Time (Static Profiling)

Just as with dynamic profiling, static profiling can be used to determine which parts of a program take most of the execution time. It can provide a good alternative if you do not want that your code is affected by extra code.

Overview of steps to perform

To obtain a profile using code instrumentation, perform the following steps:

1. Compile and link your program with static profiling enabled

2. Display the profile

First you need a completed project. If you are not using your own project, use the `profiling` example as described in Section 6.2, *Profiling using Code Instrumentation (Dynamic Profiling)*.

6.3.1. Step 1: Build your Application with Static Profiling

The first step is to tell the C compiler to make an estimation of the profiling information of your application. This is done with C compiler options:

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

The Properties for profiling dialog box appears.

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

     In the right pane the Settings appear.

3.   On the Tool Settings tab, expand the **C/C++ Compiler** entry and select **Debugging**.

4.   Enable **Static profiling**.

     For the command line, see the C compiler option **--profile** (**-p**).

     Profiling is only possible with optimization levels 0, 1 and 2. So:

5.   Open the **Optimization** page and set the **Optimization level** to **2 - Optimize more**.

6.   Click **OK** to apply the new option settings and rebuild the project (🖼).

6.3.2. Step 2: Displaying Static Profiling Results

After your project has been built with static profiling, the result of the profiler can be displayed in the TASKING Profiler perspective. The profiling data of each individual file (.sxml), is combined in the XML file profiling.xprof. This file is read and its information is displayed. To view the profiling information, open the TASKING Profiler perspective:

1.   From the **Window** menu, select **Open Perspective** » **Other...**

     The Select Perspective dialog appears.

2.   Select the **TASKING Profiler** perspective and click **OK**.

     The TASKING Profiler perspective opens. This perspective is explained in Section 6.2.3, Step 3: Displaying Profiling Results

To display static profiling information in the Profiler view

1.   In the Profiler view, click on the 📂 (Select Profiling File(s)) button.

     The Select Profiling File(s) dialog appears.

2.   In the **Profiling Type** group box, select **Static Profiling**.

3.   In the **Static Profiling File** group box, enable the option **Use default**.

     By default, the file *project*.xprof is used (profiling.xprof). If you want to specify another file, disable the option **Use default** and use the edit field and/or Browse button to specify a static profiling file (.xprof).

4.   Click **OK** to finish.

Chapter 7. Using the Assembler

This chapter describes the assembly process and explains how to call the assembler.

The assembler converts hand-written or compiler-generated assembly language programs into machine language, resulting in object files in the ELF/DWARF object format.

The assembler takes the following files for input and output:
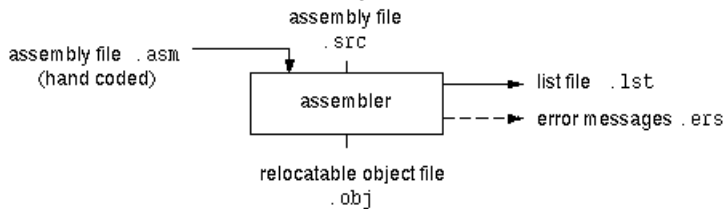
The following information is described:

• 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 12.4, *Assembler Options*.

• The various assembler optimizations.

• How to generate a list file.

• Types of assembler messages.

7.1. Assembly Process

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

Phases of the assembly process

• Parsing of the source file: preprocessing of assembler directives and checking of the syntax of instructions

• Optimization (instruction size and generic instructions)

• Generation of the relocatable object file and optionally a list file

The assembler integrates file inclusion and macro facilities. See Section 3.10, *Macro Operations* for more information.

7.2. Calling the Assembler

C166 under Eclipse can use the internal builder (default) or the TASKING makefile generator (external builder) to build your entire project. After you have built your project, the output files are available in a

245

subdirectory of your project directory, depending on the active configuration you have set in the **C/C++ Build » Settings** page of the **Project » Properties for** dialog.

Building a project under Eclipse

You have several ways of building your project:

- Build Selected File(s) (⚏). This compiles and assembles the selected file(s) without calling the linker.

  1. In the C/C++ Projects view, select the files you want to compile.

  2. Right-click in the C/C++ Projects view and select **Build Selected File(s)**.

- Build Individual Project (⚏).

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

- Rebuild Project (⚏). This builds every file in the project whether or not a file has been modified since the last build. A rebuild is a clean followed by a build.

  1. Select **Project » Clean...**

  2. Enable the option **Start a build immediately** and click **OK**.

- Build Automatically. This performs a build of all projects whenever any project file is saved, such as your makefile.

  This way of building is not recommended for C/C++ development, but to enable this feature select **Project » Build Automatically** and ensure there is a check mark beside the **Build Automatically** menu item. In order for this option to work, you must also enable option **Build on resource save (Auto build)** on the **Behaviour** tab of the **C/C++ Build** page of the **Project » Properties for** dialog.

Select a target processor (core)

Processor options affect the invocation of all tools in the toolset. In Eclipse you only need to set them once. Based on the target processor, the compiler includes a special function register file. This is a regular include file which enables you to use virtual registers that are located in memory.

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

   The Properties dialog appears.

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

   In the right pane the Processor page appears.

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

To access the assembler options

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

The Properties dialog appears.

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

   In the right pane the Settings appear.

3. On the Tool Settings tab, select **Assembler**.

4. Select the sub-entries and set the options in the various pages.

> Note that the options you enter in the Assembler page are not only used for hand-coded assembly files, but also for the assembly files generated by the compiler.

You can find a detailed description of all assembler options in Section 12.4, *Assembler Options*.

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

as166 [[*option*]... [*file*]...]...

The input file must be an assembly source file (.asm or .src).

7.3. How the Assembler Searches Include Files

When you use include files (with the .INCLUDE directive), you can specify their location in several ways. The assembler searches the specified locations in the following order:

1. If the .INCLUDE directive contains an absolute path name, the assembler looks for this file. If no path or a relative path is specified, the assembler looks in the same directory as the source file.

2. When the assembler did not find the include file, it looks in the directories that are specified in the **Assembler » Include Paths** page in the **C/C++ Build » Settings » Tool Settings** tab of the Project Properties dialog (equivalent to the **-I** command line option).

3. When the assembler did not find the include file (because it is not in the specified include directory or because no directory is specified), it looks in the path(s) specified in the environment variable AS166INC.

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:

as166 -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`AS166INC` and then in the default `include` directory.

7.4. Assembler Optimizations

The assembler can perform various optimizations that you can enable or disable.

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

   The Properties dialog appears.

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

   In the right pane the Settings appear.

3. On the Tool Settings tab, select **Assembler » Optimization**.

4. Enable one or more optimizations.

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

Allow JMPA+/JMPA- for speed optimizations (option -Oa/-OA)

This optimization is only available for XC16x and Super10 targets. When this option is enabled, the generic instructions JMP, JMP+, JMP-, JB+ and JB- can lead to an optimized JMPA+ or JMPA- instruction. When this optimization is disabled, JMPR is used in all situations. This leads to a smaller code size. By default this option is enabled.

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. If you turn off this optimization, generic instructions are not allowed. In that case you have to use hardware instructions.

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.

7.5. Generating a List File

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

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

To generate a list file

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

     The Properties dialog appears.

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

     In the right pane the Settings appear.

3.   On the Tool Settings tab, select **Assembler » List File**.

4.   Enable the option **Generate list file**.

5.   (Optional) Enable the options to include that information in the list file.

Example on the command line (Windows Command Prompt)

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

```
as166 -l test.asm
```

See Section 14.1, *Assembler List File Format*, for an explanation of the format of the list file.

7.6. Assembler Error Messages

The assembler reports the following types of error messages in the Problems view of Eclipse.

F (Fatal errors)

After a fatal error the assembler immediately aborts the assembly process.

E (Errors)

Errors are reported, but the assembler continues assembling. No output files are produced unless you have set the assembler option **--keep-output-files** (the resulting output file may be incomplete).

W (Warnings)

Warning messages do not result into an erroneous assembly output file. They are meant to draw your attention to assumptions of the assembler for a situation which may not be correct. You can control warnings in the **C/C++ Build » Settings » Tool Settings » Assembler » Diagnostics** page of the **Project » Properties for** menu (assembler option **--no-warnings**).

Display detailed information on diagnostics

1. From the **Window** menu, select **Show View » Other » General » Problems**.

    The Problems view is added to the current perspective.

2. In the Problems view right-click on a message.

    A popup menu appears.

3. Select **Detailed Diagnostics Info**.

    A dialog box appears with additional information.

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

```
as166 --diag=[format:]{all | number,...]
```

Chapter 8. Using the Linker

This chapter describes the linking process, how to call the linker and how to control the linker with a script file.

The TASKING linker is a combined linker/locator. The linker phase combines relocatable object files (.obj files, generated by the assembler), and libraries into a single relocatable linker object file (.out). The locator phase assigns absolute addresses to the linker object file and creates an absolute object file which you can load into a target processor. From this point the term linker is used for the combined linker/locator.

The linker can simultaneously link and locate all programs for all cores available on a target board. The target board may be of arbitrary complexity. A simple target board may contain one standard processor with some external memory that executes one task. A complex target board may contain multiple standard processors and DSPs combined with configurable IP-cores loaded in an FPGA. Each core may execute a different program, and external memory may be shared by multiple cores.

The linker takes the following files for input and output:
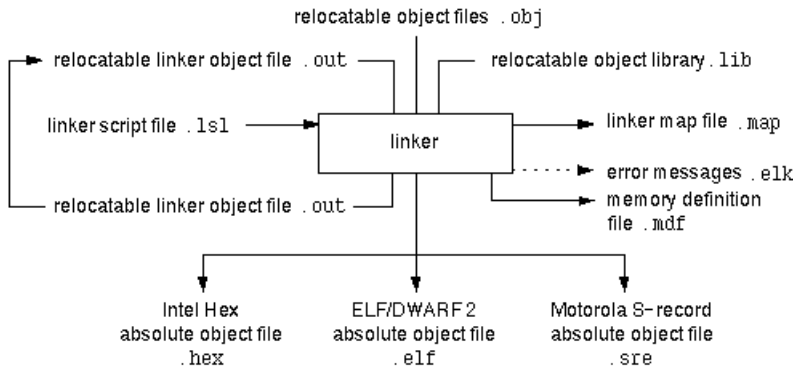
This chapter first describes the linking process. Then it describes how to call the linker and how to use its options. An extensive list of all options and their descriptions is included in Section 12.5, *Linker Options*.

To control the link process, you can write a script for the linker. This chapter shortly describes the purpose and basic principles of the Linker Script Language (LSL) on the basis of an example. A complete description of the LSL is included in Linker Script Language.

8.1. Linking Process

The linker combines and transforms relocatable object files (.obj) into a single absolute object file. This process consists of two phases: the linking phase and the locating phase.

In the first phase the linker combines the supplied relocatable object files and libraries into a single relocatable object file. In the second phase, the linker assigns absolute addresses to the object file so it can actually be loaded into a target.

Terms used in the linking process

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 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
	• 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.

8.1.1. Phase 1: Linking

The linker takes one or more relocatable object files and/or libraries as input. A relocatable object file, as generated by the assembler, contains the following information:

* *Header information*: Overall information about the file, such as the code size, name of the source file it was assembled from, and creation date.

* *Object code*: Binary code and data, divided into various named sections. Sections are contiguous chunks of code that have to be placed in specific parts of the memory. The program addresses start at zero for each section in the object file.

* *Symbols*: Some symbols are exported - defined within the file for use in other files. Other symbols are imported - used in the file but not defined (external symbols). Generally these symbols are names of routines or names of data objects.

* *Relocation information*: A list of places with symbolic references that the linker has to replace with actual addresses. When in the code an external symbol (a symbol defined in another file or in a library) is referenced, the assembler does not know the symbol's size and address. Instead, the assembler generates a call to a preliminary relocatable address (usually 0000), while stating the symbol name.

* *Debug information*: Other information about the object code that is used by a debugger. The assembler optionally generates this information and can consist of line numbers, C source code, local symbols and descriptions of data structures.

The linker resolves the external references between the supplied relocatable object files and/or libraries and combines the files into a single relocatable linker object file.

The linker starts its task by scanning all specified relocatable object files and libraries. If the linker encounters an unresolved symbol, it remembers its name and continues scanning. The symbol may be defined elsewhere in the same file, or in one of the other files or libraries that you specified to the linker. If the symbol is defined in a library, the linker extracts the object file with the symbol definition from the library. This way the linker collects all definitions and references of all of the symbols.

Next, the linker combines sections with the same section name and attributes into single sections. The linker also substitutes (external) symbol references by (relocatable) numerical addresses where possible.

At the end of the linking phase, the linker either writes the results to a file (a single relocatable object file) or keeps the results in memory for further processing during the locating phase.

The resulting file of the linking phase is a single relocatable object file (`.out`). If this file contains unresolved references, you can link this file with other relocatable object files (`.obj`) or libraries (`.lib`) to resolve the remaining unresolved references.

With the linker command line option **--link-only**, you can tell the linker to only perform this linking phase and skip the locating phase. The linker complains if any unresolved references are left.

8.1.2. Phase 2: Locating

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

Code modification

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

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

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

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

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

Output formats

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

Controlling the linker

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

• The memory installed in the embedded target system:

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

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

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

- The underlying hardware architecture of the target processor.

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

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

See also Section 8.7, *Controlling the Linker with a Script*.

8.2. Calling the Linker

In Eclipse you can set options specific for the linker. After you have built your project, the output files are available in a subdirectory of your project directory, depending on the active configuration you have set in the **C/C++ Build » Settings** page of the **Project » Properties for** dialog.

Building a project under Eclipse

You have several ways of building your project:

- Build Individual Project ().

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

- Rebuild Project (). This builds every file in the project whether or not a file has been modified since the last build. A rebuild is a clean followed by a build.

  1. Select **Project » Clean...**

  2. Enable the option **Start a build immediately** and click **OK**.

- Build Automatically. This performs a build of all projects whenever any project file is saved, such as your makefile.

This way of building is not recommended, but to enable this feature select **Project » Build Automatically** and ensure there is a check mark beside the **Build Automatically** menu item. In order for this option to work, you must also enable option **Build on resource save (Auto build)** on the **Behaviour** tab of the **C/C++ Build** page of the **Project » Properties for** dialog.

To access the linker options

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

    The Properties dialog appears.

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

    In the right pane the Settings appear.

3. On the Tool Settings tab, select **Linker**.

4. Select the sub-entries and set the options in the various pages.

You can find a detailed description of all linker options in Section 12.5, *Linker Options*.

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

lk166 [[*option*]... [*file*]...]...

When you are linking multiple files, either relocatable object files (.obj) or libraries (.lib), it is important to specify the files in the right order. This is explained in Section 8.3, *Linking with Libraries*.

Example:

```
lk166 -dxc16x.lsl test.obj
```

This links and locates the file test.obj and generates the file test.elf.

8.3. Linking with Libraries

There are two kinds of libraries: system libraries and user libraries.

System library

System libraries are stored in the directories:

```
<C166 installation path>\lib\[p]1 (c16x/st10/st10mac libraries)
<C166 installation path>\lib\2    (xc16x/super10/super10m345 libraries)
```

The p1 directory contains the protected libraries for CPU functional problems.

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

Libraries	Description
c166c*m*[n][u][s].lib	C libraries for each model *m*: n (near), f (far), s (shuge), h (huge) Optional letters: n = near functions u = user stack s = single precision floating-point
c166fp*m*[n][u][t].lib	Floating-point libraries for each model *m*: n, f, s, h Optional letters: n = near functions u = user stack t = trapping
c166rt*m*[n][u].lib	Run-time libraries for each model *m*: n, f, s, h Optional letters: n = near functions u = user stack
c166pb*m*[n][u].lib c166pc*m*[n][u].lib c166pct*m*[n][u].lib c166pd*m*[n][u].lib c166pt*m*[n][u].lib	Profiling libraries for each model *m*: n, f, s, h pb = block/function counter pc = call graph pct = call graph and timing pd = dummy pt = function timing Optional letters: n = near functions u = user stack
c166cp*m*[u][x].lib	C++ libraries for each model *m*: n, f, s, h Optional letters: u = user stack x = exception handling
c166stl*m*[u]x.lib [*]	STLport C++ libraries for each model *m*: n, f, s, h Optional letters: u = user stack x = exception handling

[*] From the STLport C++ library only the near model variant is delivered ready-to-use. The other STLport C++ libraries are delivered in source. You can build them yourself when you need them. See Chapter 13, *Libraries* for more information.

To link the default C (system) libraries

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

   The Properties dialog appears.

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

   In the right pane the Settings appear.

3.   On the Tool Settings tab, select **Linker » Libraries**.

4.   Enable the option **Link default libraries**.

5.   Enable or disable the option **Use trapped floating-point library**.

When you want to link system libraries from the command line, you must specify this with the option **--library** (**-l**). For example, to specify the system library `c166cf.lib`, type:

```
lk166 --library=cf test.obj
```

User library

You can create your own libraries. Section 9.4, *Archiver* describes how you can use the archiver to create your own library with object modules.

To link user libraries

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

     The Properties dialog appears.

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

     In the right pane the Settings appear.

3.   On the Tool Settings tab, select **Linker » Libraries**.

4.   Add your libraries to the **Libraries** box.

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

```
lk166 start.obj mylib.lib
```

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

```
lk166 start.obj mylibs\mylib.lib
```

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

Library order

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

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

```
lk166 --first-library-first a.lib test.obj b.lib
```

If the file `test.obj` calls a function which is both present in `a.lib` and `b.lib`, normally the function in `b.lib` would be extracted. With this option the linker first tries to extract the symbol from the first library `a.lib`.

Note that routines in `b.lib` that call other routines that are present in both `a.lib` and `b.lib` are now also resolved from `a.lib`.

8.3.1. How the Linker Searches Libraries

System libraries

You can specify the location of system libraries in several ways. The linker searches the specified locations in the following order:

1. The linker first looks in the **Library search path** that are specified in the **Linker » Libraries** page in the **C/C++ Build » Settings » Tool Settings** tab of the Project Properties dialog (equivalent to the **-L** command line option). If you specify the **-L** option without a pathname, the linker stops searching after this step.

2. When the linker did not find the library (because it is not in the specified library directory or because no directory is specified), it looks in the path(s) specified in the environment variable `LIBC166`.

3. When the linker did not find the library, it tries the default `lib` directory relative to the installation directory.

User library

If you use your own library, the linker searches the library in the current directory only.

8.3.2. How the Linker Extracts Objects from Libraries

A library built with the TASKING archiver **ar166** 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:

```
lk166 mylib.lib
```

nothing is linked and no output file will be produced, because there are no unresolved symbols when the linker searches through `mylib.lib`.

It is possible to force a symbol as external (unresolved symbol) with the option **--extern** (**-e**):

```
lk166 --extern=main mylib.lib
```

In this case the linker searches for the symbol `main` in the library and (if found) extracts the object that contains `main`.

If this module contains new unresolved symbols, the linker looks again in `mylib.lib`. This process repeats until no new unresolved symbols are found.

8.4. Incremental Linking

With the TASKING linker it is possible to link incrementally. Incremental linking means that you link some, but not all `.obj` modules to a relocatable object file `.out`. In this case the linker does not perform the locating phase. With the second invocation, you specify both new `.obj` files as the `.out` file you had created with the first invocation.

Incremental linking is only possible on the command line.

```
lk166 -dxc16x.lsl --incremental test1.obj -otest.out
lk166 -dxc16x.lsl test2.obj test.out
```

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

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

8.5. Importing Binary Files

With the TASKING linker it is possible to add a binary file to your absolute output file. In an embedded application you usually do not have a file system where you can get your data from. With the linker option **--import-object** you can add raw data to your application. This makes it possible for example to display images on a device or play audio. The linker puts the raw data from the binary file in a section. The section is aligned on a 2-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>
__huge extern char    _lc_ub_my_mp3; /* linker labels */
__huge extern char    _lc_ue_my_mp3;
```

```
__huge 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 type qualifier `__huge`, otherwise the linker cannot bind
your linker symbols.

Also note that if you want to use the export functionality of Eclipse, the binary file has to be part
of your project.

8.6. Linker Optimizations

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

To enable or disable optimizations

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

    The Properties dialog appears.

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

    In the right pane the Settings appear.

3. On the Tool Settings tab, select **Linker » Optimization**.

4. Enable one or more optimizations.

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

Delete unreferenced sections (option -Oc/-OC)

This optimization removes unused sections from the resulting object file.

First fit decreasing (option -Ol/-OL)

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

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

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

Compress copy table (option -Ot/-OT)

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

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

Delete duplicate code (option -Ox/-OX)

Delete duplicate constant data (option -Oy/-OY)

These two optimizations remove code and constant data that is defined more than once, from the resulting object file.

Compress ROM sections of copy table items (option -Oz/-OZ)

Reduces the size of the application's ROM image by compressing the ROM image of initialized data sections. At application startup time the ROM image is decompressed and copied to RAM.

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

8.7. Controlling the Linker with a Script

With the options on the command line you can control the linker's behavior to a certain degree. From Eclipse it is also possible to determine where your sections will be located, how much memory is available, which sorts of memory are available, and so on. Eclipse passes these locating directions to the linker via a script file. If you want even more control over the locating process you can supply your own script.

The language for the script is called the *Linker Script Language*, or shortly LSL. You can specify the script file to the linker, which reads it and locates your application exactly as defined in the script. If you do not specify your own script file, the linker always reads a standard script file which is supplied with the toolset.

8.7.1. Purpose of the Linker Script Language

The Linker Script Language (LSL) serves three purposes:

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

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

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

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

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

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

The complete LSL syntax is described in Chapter 16, *Linker Script Language (LSL)*.

8.7.2. Eclipse and LSL

In Eclipse you can specify the size of the stack and heap; the physical memory attached to the processor; identify that particular address ranges are reserved; and specify which sections are located where in memory. Eclipse translates your input into an LSL file that is stored in the project directory under the name *project_name*.`lsl` and passes this file to the linker. If you want to learn more about LSL you can inspect the generated file *project_name*.`lsl`.

To add a generated Linker Script File to your project

1. From the **File** menu, select **File » New » TASKING VX-toolset for C166 C/C++ Project**.

   The New C/C++ Project wizard appears.

2. Fill in the project settings in each dialog and click **Next >** until the following dialog appears.
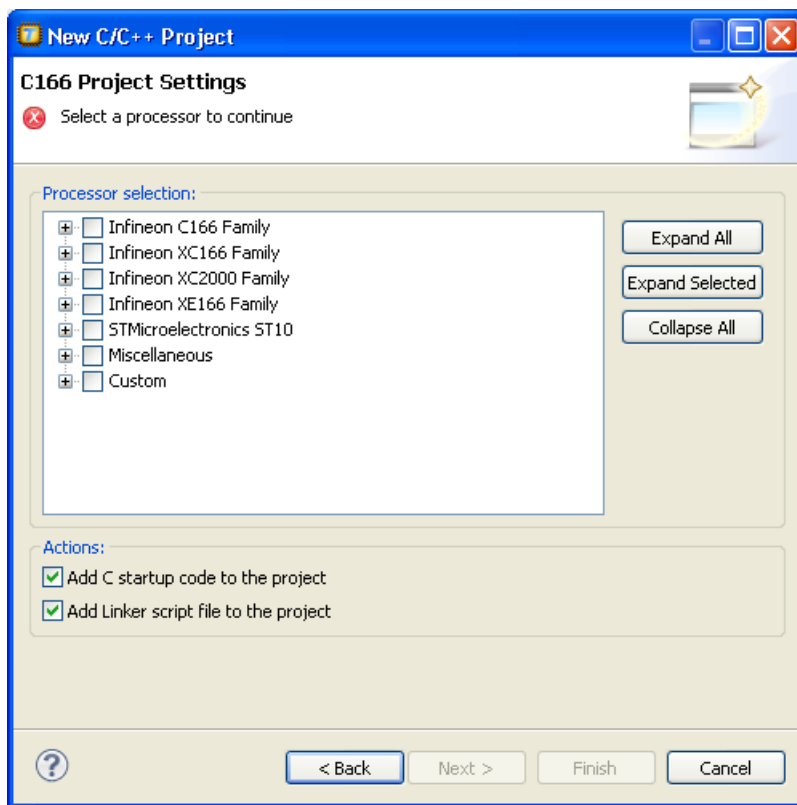
3. Enable the option **Add Linker script file to the project** and click **Finish**.

    Eclipse creates your project and the file "project_name.lsl" in the project directory.
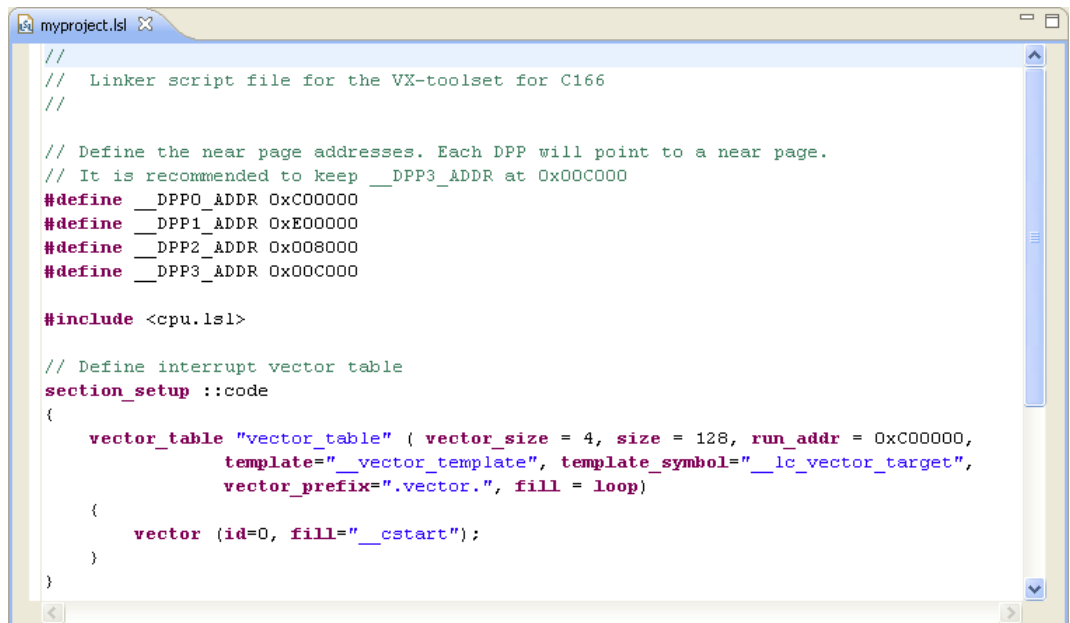
If you do not add the linker script file here, you can always add it later with **File » New » Linker Script File (LSL)**.

To change the Linker Script File in Eclipse

There are two ways of changing the LSL file in Eclipse.

*   You can change the LSL file directly in an editor.

    1. Double-click on the file *project_name*.lsl.

        The project LSL file opens in the editor area.

2. You can edit the LSL file directly in the *project_name*.lsl editor.

    *A * appears in front of the name of the LSL file to indicate that the file has changes.*

3. Click 💾 or select **File » Save** to save the changes.

• You can also make changes to the property pages Memory, Stack/Heap and Vector Table.

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

        The Properties dialog appears.

    2. In the left pane, expand **C/C++ Build** and select **Memory**, or **Stack/Heap** or **Vector Table**.

        In the right pane the corresponding property page appears.

    3. Make changes to memory, stack/heap and/or vector table and click **OK**.

        The project LSL file is updated automatically according to the changes you make in the pages.

You can quickly navigate through the LSL file by using the Outline view (**Window » Show View » Outline**).

8.7.3. Structure of a Linker Script File

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

The architecture definition (required)

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

This specification is normally written by Altium. Altium supplies LSL files in the `include.lsl` directory. The file `arch_c166.lsl` defines the base architecture for all cores. The files `arch_core.lsl` extend the base architecture for *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 common derivatives for each core in the files `arch_core.lsl` and supplies LSL files for each derivative (*derivative*`.lsl`) which extends the common derivative.

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

```
architecture c166
{
    // Specification of the c166 core architecture.
    // Written by Altium.
}

architecture xc16x extends c166
{   // xc16x inherits all features of c166
}

derivative X  // derivative name is arbitrary
{
    // Specification of the derivative.
    // Written by Altium.
    core xc16x          // always specify the core
    {
        architecture = xc16x;
    }

    bus address        // internal bus
    {
        // maps to bus "address" in "xc16x" core
    }

    // internal memory
}

processor spe        // processor name is arbitrary
{
```

```
    derivative = X;

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

memory ext_name
{
    // external memory definition
}

section_layout spe:xc16x:shuge     // section layout
{
    // section placement statements

    // sections are located in address space 'shuge'
    // of core 'xc16x' of processor 'spe'
}
```

Overview of LSL files delivered by Altium

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

LSL file	Description
`arch_c166.lsl`	Defines the base architecture for all cores.
`arch_core.lsl`	Extends the base architecture for *core* and defines a common derivative. It includes the file `arch_c166.lsl`.
`derivative.lsl`	Extends the common derivative as defined for *core* and defines a single processor. It includes the file `arch_core.lsl`.
`template.lsl`	This file is used by Eclipse as a template for the project LSL file. It includes the file `derivative.lsl` based on your CPU selection. The CPU is specified by the __CPU__ macro.
`default.lsl`	Contains a default memory definition and section layout. It includes the file `derivative.lsl` based on your CPU selection. The CPU is specified by the __CPU__ macro. If you invoke the control program without a CPU specification, the linker is called with **-D__CPU__=c16x**.

When you select to add a linker script file when you create a project in Eclipse, Eclipse makes a copy of the file `template.lsl` and names it "*project_name*.lsl". On the command line, the linker uses the file `default.lsl`, unless you specify another file with the linker option **--lsl-file** (**-d**).

8.7.4. The Architecture Definition

Although you will probably not need to program the architecture definition (unless you are building your own processor core) it helps to understand the Linker Script Language and how the definitions are interrelated.

Within an *architecture definition* the characteristics of a target processor core that are important for the linking process are defined. These include:

- space definitions: the logical address spaces and their properties

- bus definitions: the I/O buses of the core architecture

- mappings: the address translations between logical address spaces, the connections between logical address spaces and buses and the address translations between buses

Address spaces

A logical address space is a memory range for which the core has a separate way to encode an address into instructions. Most microcontrollers and DSPs support multiple address spaces. For example, 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 `c166` as defined in `arch_c166.lsl`.

Space	Id	MAU	Description
bit	1	1	Select all bit sections. This space starts at address 0xFD00 and is 0x800 bits long, ending at 0xFDFF.
bita	2	8	Bit-addressable space.
iram	3	8	Internal memory, usually Dual Port RAM. The size is always 3 kB and ranges from 0xF200 to 0xFDFF.
near	4	8	Near data space, 4 16 kB pages anywhere in memory. All four DPPs each point to one of these 4 pages. DPP3 is fixed to page 3 (0xC000) to facilitate access of SFRs through the MEM addressing mode. DPP0, DPP1 and DPP2 can be assigned to any page in memory. It also defines a user stack and a heap (`nheap`).
far	5	8	Far data, also used for grouped 'SYSTEM' sections.
shuge	6	8	Segmented huge data, contains definitions for copy table and system stack.
huge	7	8	Huge data, also defines a heap (`hheap`) and linker symbols.
code	8	8	Code address space, specifies the start address at the beginning of the vector table.

By default the near space is 'paged' in pages of 16 kB. The first byte in each space is reserved to avoid NULL pointer comparison problems with objects allocated at the beginning of the page. It is possible to

remove the page restriction in the near space by defining the `__CONTIGUOUS_NEAR` macro. This makes it possible to allocate objects larger that 16 kB or to make a user stack larger than 16 kB. But with this page restriction removed, you should not cast a near to a far or shuge pointer in C, unless you are absolutely sure that the section of the object pointed to does not cross a page boundary.

The spaces are nested in such a way that the locate algorithm uses the right order. The linker starts with locating the sections that are most far away from the bus definition, which means that sections for spaces with the highest memory range restriction will be located first. The following space nesting is used:

```
bus:address
 |
 \---huge
      |
      |---code
      |
      \---shuge
           |
           \---far
                |
                \---near
                     |
                     \---iram
                          |
                          \---bita
                               |
                               \---bit
```
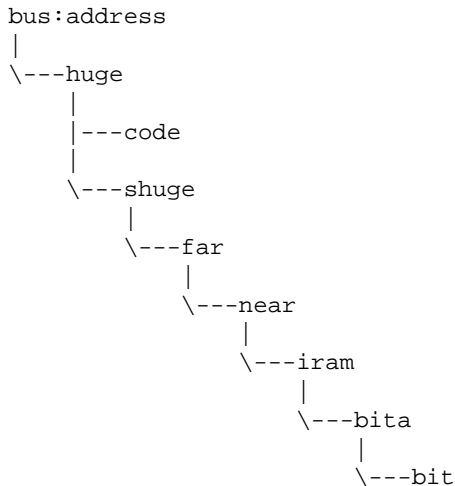
The C166 architecture in LSL notation

The best way to program the architecture definition, is to start with a drawing. The figure below shows a part of the architecture `c166`, it shows an example of how the near linear pages are mapped into memory.
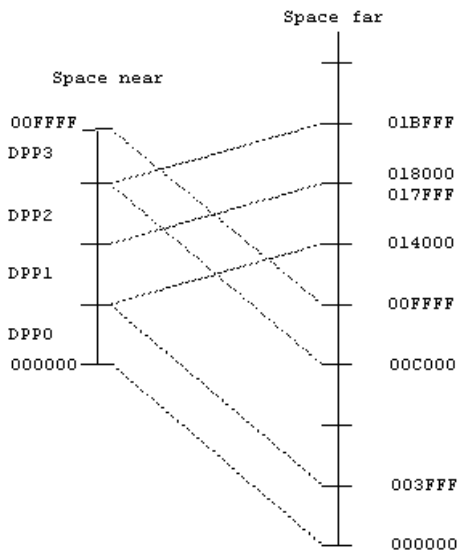
The figure shows two address spaces called `near` and `far`. All address spaces have attributes like a number that identifies the logical space (id), a MAU and an alignment. In LSL notation the definition of these address spaces looks as follows:

```
#define __DPP0_ADDR 0x000000
#define __DPP1_ADDR 0x014000
#define __DPP2_ADDR 0x018000

space near
{
  id  = 4;
  mau = 8;
  page;

  map( src_offset = 0x0000, dest_offset = __DPP0_ADDR, dest=space:far, size=16k );
  map( src_offset = 0x4000, dest_offset = __DPP1_ADDR, dest=space:far, size=16k );
  map( src_offset = 0x8000, dest_offset = __DPP2_ADDR, dest=space:far, size=16k );
  map( src_offset = 0xC000, dest_offset = 0xC000, dest=space:far, size=16k );
  // ...
}

space far
{
  id  = 5;
  mau = 8;
  page;
  page_size = 0x4000 [__PAGE_START..0x4000 - __PAGE_END];

  map( size = 16M, dest = space:shuge );
}

space huge
{
  id  = 7;
  mau = 8;
  page_size = 0x1000000 [__PAGE_START..0x1000000 - __PAGE_END];

  map( size = 16M, dest = bus:address );
}
```

The keyword `map` corresponds with the dotted lines in the drawing. You can map:

- address space => address space

- address space => bus (not shown in the drawing)

- memory => bus (not shown in the drawing)

- bus => bus (not shown in the drawing)

Next the internal bus named `address` must be defined in LSL:

```
bus address
{
    mau = 8;
    width = 24;  // there are 24 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 c166
{
    // All code above goes here.
}
```

8.7.5. The Derivative Definition

Although you will probably not need to program the derivative definition (unless you are using multiple cores) it helps to understand the Linker Script Language and how the definitions are interrelated.

A *derivative* is the design of a processor, as implemented on a chip (or FPGA). It comprises one or more cores and on-chip memory. The derivative definition includes:

• core definition: an instance of a core architecture

• bus definition: the I/O buses of the core architecture

• memory definitions: internal (or on-chip) memory (in Eclipse this is called 'System memory')

Core

Each derivative must have at least one core and each core must have a specification of its core architecture. This core architecture must be defined somewhere in the LSL file(s).

```
core xc16x
{
    architecture = xc16x;
}
```

Bus

Each derivative can contain a bus definition for connecting external memory. In this example, the bus address maps to the bus address defined in the architecture definition of core xc16x:

```
bus address
{
    mau   = 8;
    width = 24;
    map( dest=bus:xc16x:address, dest_offset=0, size=16M );
}
```

Memory

Memory is usually described in a separate memory definition, but you can specify on-chip memory for a derivative. For example:

```
memory dpram
{
    mau  = 8;
    type = ram;
    size = 2k;
    map( dest=bus:xc16x:address, src_offset = 0, dest_offset = 0xF600, size = 2k );
}
```

This completes the LSL code in the derivative definition. Note that all code above goes into the derivative definition, thus between:

```
derivative X    // name of derivative
{
    // All code above goes here
}
```

8.7.6. The Processor Definition

The processor definition is only needed when you write an LSL file for a multi-processor embedded system. The processor definition explicitly instantiates a derivative, allowing multiple processors of the same type.

```
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 = xc2287m;
}
```

8.7.7. The Memory Definition

Once the core architecture is defined in LSL, you may want to extend the processor with 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 128 kB of external ROM, named `xrom`, 64 kB of external RAM, named `xram` and 16 kB of external NVRAM, named `my_nvram`. All memories are connected to the bus `address`. In LSL this looks like:

```
memory my_nvram
{
    mau  = 8;
    type = nvram;
    size = 16k;
    map( dest=bus:spe:address, src_offset = 0, dest_offset = 16k, size = 16k );
}

memory xrom
{
    mau  = 8;
    type = rom;
    size = 128k;
    map( dest=bus:spe:address, src_offset = 0, dest_offset = 64k, size = 128k );
}

memory xram
{
    mau  = 8;
    type = ram;
    size = 64k;
    map( dest=bus:spe:address, src_offset = 0, dest_offset = 192k, size = 64k );
}
```

If you use a different memory layout than described in the LSL file supplied for the target core, you can specify this in Eclipse or you can specify this in a separate LSL file and pass both the LSL file that describes the core architecture and your LSL file that contains the memory specification to the linker.

To add memory using Eclipse

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

    The Properties dialog appears.

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

    In the right pane the Memory page appears.

3. Open the **Memory** tab and click on the **Add...** button.

    The Add new memory dialog appears.

4. Enter the memory name (for example `my_nvram`), type (for example `nvram`) and size.

5. Click on the **Add...** button.

The Add new mapping dialog appears.

6. You have to specify at least one mapping. Enter the mapping name (optional), address, size and destination and click **OK**.

   The new mapping is added to the list of mappings.

7. Click **OK**.

   The new memory is added to the list of memories (user memory).

8. Click **OK** to close the Properties dialog.

   The updated settings are stored in the project LSL file.

If you make changes to the on-chip memory as defined in the architecture LSL file, the memory is copied to your project LSL file and the line `#define __REDEFINE_ON_CHIP_ITEMS` is added. If you remove all the on-chip memory from your project LSL file, also make sure you remove this define.

8.7.8. The Section Layout Definition: Locating Sections

Once you have defined the internal core architecture and optional memory, you can actually define where your application must be located in the physical memory.

During compilation, the compiler divides the application into sections. Sections have a name, an indication (section type) in which address space it should be located and attributes like writable or read-only.

In the section layout definition you can exactly define how input sections are placed in address spaces, relative to each other, and what their absolute run-time and load-time addresses will be.

Example: section propagation through the toolset

To illustrate section placement, the following example of a C program (`bat.c`) is used. The program saves the number of times it has been executed in battery back-upped memory, and prints the number.

```
#define BATTERY_BACKUP_TAG  0xa5f0
#include <stdio.h>

int  uninitialized_data;
int  initialized_data = 1;
#pragma section near=non_volatile
#pragma noclear
int  battery_backup_tag;
int  battery_backup_invok;
#pragma clear
#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` is defined. In this section the battery back-upped data is stored.

By default the compiler creates a section with the name "near" of section type "near" carrying section attributes "clear" and "new" to store uninitialized data objects. The section type and attributes tell the linker to locate the section in address space near and that the section content should be filled with zeros at startup.

As a result of the `#pragma section near=non_volatile`, the data objects between the pragma pair are placed in a section with the name "non_volatile". Note that the compiler sets the "clear" attribute. However, battery back-upped sections should not be cleared and therefore we used `#pragma noclear`.

Section placement

The number of invocations of the example program should be saved in non-volatile (battery back-upped) memory. This is the memory `my_nvram` from the example in Section 8.7.7, *The Memory Definition*.

To control the locating of sections, you need to write one or more section definitions in the LSL file. At least one for each address space where you want to change the default behavior of the linker. In our example, we need to locate sections in the address space `near`:

```
section_layout ::near
{
    // 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 `non_volatile`. All other sections are located using the defaults specified in the architecture definition. Section `non_volatile` should be placed in non-volatile ram. To achieve this, the run address refers to our non-volatile memory called `my_nvram`.

```
group ( ordered, run_addr = mem:my_nvram )
{
    select "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.

8.7.9. Copying Code Sections to PSRAM at Startup

For fast performance or preparing power-down modes it could be necessary to copy some (critical) code portions from flash memory to PSRAM. You can do this easily from the LSL file by using the `copy` attribute.

```
section_layout ::code
{
    group( ordered, copy )
    {
        select "code_libc";
    }
}
```

This creates a ROM copy of all sections named "`code_libc`". For each selected section a ROM section and a RAM section is created. The ROM section is named "`[code_libc]`", while the RAM section is named "`code_libc`". For these sections entries will be created in the copy table, resulting in the contents of the ROM section (in flash) being copied to RAM.

See the tables in Section 8.7.11, *ICACHE Support and Named Memory Mappings* for an overview of which memory ranges are available for code (ROM and RAM).

8.7.10. PSRAM Mirrors

All XC2000/XE166 devices have on-chip Program SRAM (PSRAM), starting at address 0xE00000. A mirror of the PSRAM is always located starting at address 0xE80000. This mirror has timing parameters that correspond to flash timing. This mirror is therefore called Emulated PSRAM (EPSRAM). PSRAM and EPSRAM both support code and data access.

On devices with an ICACHE there is also a second mirror of PSRAM, starting at 0xA00000. This mirror only supports code access.

In the device specific LSL file in the product's `include.lsl` directory, and on the Eclipse Memory properties page you will see only a single PSRAM memory. This definition includes a mappings for each mirrored area. For a device with ICACHE the LSL memory definition may look like this:

```
memory PSRAM
{
    mau       = 8;
    type      = ram;
    size      = 64k;
    priority = 1;
    map max_speed    ( dest=bus:xc16x:address, dest_offset = 0xE00000,
                         size = 64k );
    map flash_timing ( dest=bus:xc16x:address, dest_offset = 0xE80000,
                         size = 64k, reserved );
    map mirror       ( dest=bus:xc16x:address, dest_offset = 0xA00000,
                         size = 64k, reserved );
}
```

The mirrors at 0xE80000 and 0xA00000 are reserved. This means that by default no sections will be located in these areas. Only absolute sections and sections with a fixed memory range can be located

in these areas. You can define an absolute section in the source file with the `__at()` attribute, or you can use an LSL `group` statement. A section can get a fixed memory range if you use an LSL `group` statement with an address range.

The addresses occupied by a section in one of the areas automatically implies that the same offsets in the mirrored area(s) are occupied.

Example:

```
section_layout ::code
{
    group( run_addr = mem:spe:PSRAM/mirror, copy )
    {
        select "code_main";
    }
}
```

In this example the `run_addr` specifies the full address range of the mirror at 0xA00000. The `copy` attribute is used to create a copy of the section in flash, which will be copied to PSRAM at startup.

When you want to use one of the mirrors as the default range, you can exchange the `reserved` keyword on the mappings. To do so, you will have to copy the device specific LSL file from the product's `include.lsl` directory to your project. Or when you use Eclipse you can simply modify the PSRAM memory definition in the **Project » Properties for » C/C++ Build » Memory » Memory** page. Eclipse will then take care of making these settings in your project LSL file.

Example:

```
memory PSRAM
{
    mau      = 8;
    type     = ram;
    size     = 64k;
    priority = 1;
    map max_speed    ( dest=bus:xc16x:address, dest_offset = 0xE00000,
                       size = 64k, reserved );
    map flash_timing ( dest=bus:xc16x:address, dest_offset = 0xE80000,
                       size = 64k, reserved );
    map mirror       ( dest=bus:xc16x:address, dest_offset = 0xA00000,
                       size = 64k );
}
```

This makes the mirror at 0xA00000 the default.

You can use EPSRAM as if it is flash by modifying the memory type from `ram` to `rom` and reserving all ranges except the `flash_timing` range:

```
memory PSRAM
{
    mau      = 8;
    type     = rom;
    size     = 64k;
```

```
    priority = 1;
    map max_speed   ( dest=bus:xc16x:address, dest_offset = 0xE00000,
                      size = 64k, reserved );
    map flash_timing ( dest=bus:xc16x:address, dest_offset = 0xE80000,
                      size = 64k );
    map mirror      ( dest=bus:xc16x:address, dest_offset = 0xA00000,
                      size = 64k, reserved );
}
```

8.7.11. ICACHE Support and Named Memory Mappings

Some of the newer Infineon devices, such as the XC2268I have a cache on the flash memory, the so-called ICACHE. To use the ICACHE it must be enabled in the C startup code (see Section 4.3.1, *ICACHE Support*) and the linker script file must be modified to get sections located in the memory ranges that are covered by the ICACHE.

Devices with ICACHE have two memory areas that are mapped to the on-chip flash. The standard flash area, starting at 0xC00000, is not cached. For code that must be fetched through the cache, you must use the area starting at 0x800000. The cached area is a mirror of the not cached area. Code located in either of these areas is available at the other area at the same offset to the start address of the area. In the cached area the processor can only execute code. No data shall be located in that area.

The standard linker script files (from the product's `include.lsl` directory) have special provisions for these areas. In general your project's LSL file includes the standard linker script file for the selected processor. This standard linker script file defines the on-chip memories, reserved areas and other processor architecture related stuff.

At the lowest level there is separation in ranges for code and data. This varies per core. The areas are listed in the following tables. These mappings are defined in the file `arch_c166.lsl` in the product's `include.lsl` directory.

XC16x, XC2xxx, XE16x devices without ICACHE

Start	End	Size	Code	Data	Description
C00000	FFFFFF	4M	x	x	Flash, PSRAM, EPSRAM
010000	BFFFFF	12M-64k	x	x	External memory and I/O
00F200	00FDFF	3k	-	x	DPRAM
008000	00DFFF	24k	-	x	DSRAM
000000	007FFF	32k	x	x	External memory

XC16x, XC2xxx, XE16x devices with ICACHE

Start	End	Size	Code	Data	Description
C00000	FFFFFF	4M	x	x	Flash, PSRAM, EPSRAM
080000	AFFFFF	3M	x	-	Flash and PSRAM mirrors
010000	7FFFFF	8M-64k	x	x	External memory and I/O

Start	End	Size	Code	Data	Description
00F200	00FDFF	3k	-	x	DPRAM
008000	00DFFF	24k	-	x	DSRAM
000000	007FFF	32k	x	x	External memory

For all other devices the full memory is mapped for code and data.

The above address mappings do not reflect the actual (on-chip) memories. The on-chip memories are defined in the device specific LSL file in the product's `include.lsl` directory. A memory definition can contain multiple named address mappings. An example of a flash memory definition for a device with ICACHE support:

```
memory Flash0
{
    mau  = 8;
    type = rom;
    size = 256k;
    map not_cached ( dest=bus:xc16x:address, dest_offset = 0xC00000,
                     size = 256k );
    map     cached ( dest=bus:xc16x:address, dest_offset = 0x800000,
                     size = 256k, reserved );
}
```

The map statements specify the addresses where this memory can be found. In the example above a mirror is defined. Both maps have the same size, but a different destination offset (`dest_offset`). The source offset (`src_offset`) is unspecified, which implies that it is 0, meaning that both maps are starting at the same position, effectively forming a mirror.

With the keyword `reserved` the map is not available for sections to be automatically located in the memory area of the map. Only 'absolute' or 'ranged' sections can then be located in the memory area specified with this map. Absolute sections can be defined in the C code (using the keyword `__at`), in assembly code (using the attribute 'at' on the `.section` directive) or in the linker script file with the `group` statement. Ranged sections are sections that are selected in the LSL file inside a group statement with an address range.

The keyword `reserved` is placed on the map of the cached area by default. If you want to place sections into the cached area, there are two options:

1. Make the sections absolute or ranged with an address in the cached area.

   For example:

   ```
   section_layout ::code
   {
       group( run_addr = mem:spe:Flash0/cached )
       {
           select "*_code";
       }
   }
   ```

The `mem:spe:Flash/cached` specification represents an address range equal to the cached map. With the `select` statement in this example all sections ending at "_code" are selected. You could also use:

```
section_layout ::code
{
    group( run_addr = mem:spe:Flash0/cached )
    {
        select "[a-z]*" (attributes=+x);
    }
}
```

The "`[a-z]*`" in the `select` statement prevents selection of the `.vector*` sections generated for the interrupt vector table. These vector sections already have an absolute address and the linker will issue warnings when trying to set a new absolute address using the group/select statement. The `+x` attribute selects all executable sections.

> Once selected with a `select` statement, sections will not be selected in a later `select` statement. So, if a code section must be placed at a special address, this must be done before the `group` statement that uses a wildcard selection.

The total flash memory of the processor consists of multiple flash arrays. In the standard LSL files each flash array has its own memory definition. In the `group` statement it is possible to specify an address range that covers multiple flash arrays. For example:

```
section_layout ::code
{
    group( run_addr = mem:spe:Flash0/cached|
                      mem:spe:Flash1/cached|
                      mem:spe:Flash2/cached )
    {
        select "[a-z]*" (attributes=+x);
    }
}
```

2. Remove the keyword `reserved` from the cached map and place it on the not_cached map.

   An alternative solution is copying the processor's specific LSL file from the product's `include.lsl` directory to your project. Then change the location of the keyword `reserved` on the map statements in the flash memory. For example:

```
memory Flash0
{
    mau  = 8;
    type = rom;
    size = 256k;
    map not_cached ( dest=bus:xc16x:address, dest_offset = 0xC00000,
                     size = 256k, reserved );
    map     cached ( dest=bus:xc16x:address, dest_offset = 0x800000,
```

```
                                    size = 256k);
}
```

Now code sections will be located in the cached map for `Flash0` by default. You need to make this modification for all flash arrays that you want to be cached.

In Eclipse you can do this in the **Edit existing memory** dialog (**Project » Properties for » C/C++ Build » Memory » Memory » Edit...**), by clicking on the **Use cached** button.
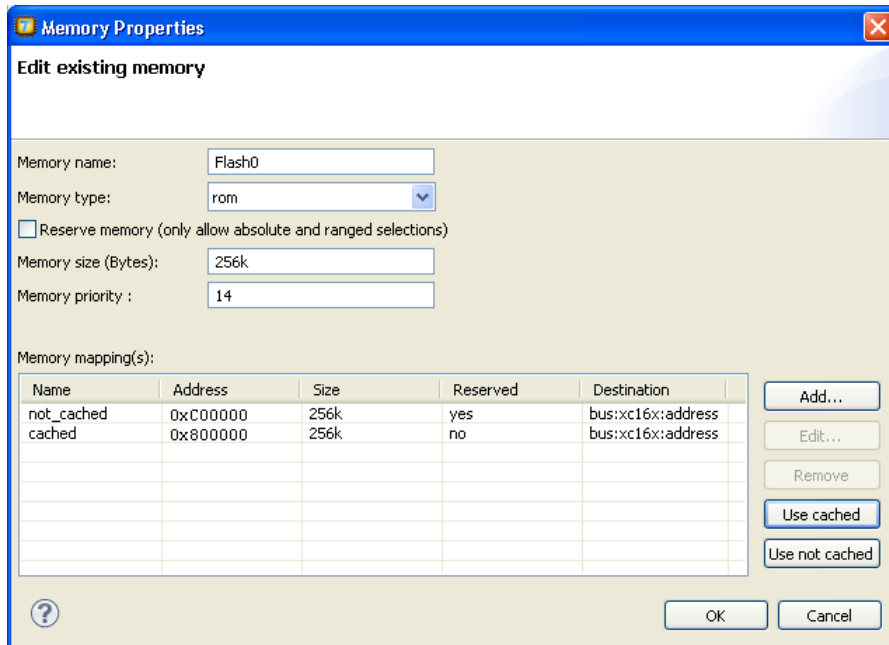
If you click the **Use not cached** button, the not cached memory is used again.

It is possible to specify per flash array what the default is, cached or not cached. The linker will then freely distribute the sections over the flash memories. If you want to control which sections are located in cached memory and which sections are located in not cached memory it is required to use the `group/select` statements. With that in mind, it is recommended to have the flash memories either all set to cached or all set to not-cached. You can then use `group/select` statements only for those sections that should not be placed in the default area.

Updating DPP addresses

When the application contains near ROM data sections, there must be at least one DPP pointing to a ROM memory. For devices with ICACHE support it is not allowed to locate data in the cached areas. The DPP must therefore point to a page in the not cached range.

The DPP addresses are defined in the project's LSL file with the macros `__DPP0_ADDR0`, `__DPP1_ADDR`, `__DPP2_ADDR` and `__DPP3_ADDR`. For example:

```
#define __DPP0_ADDR 0xC00000    // points to not cached area
#define __DPP1_ADDR 0xC04000    // points to not cached area
#define __DPP2_ADDR 0x008000
#define __DPP3_ADDR 0x00C000
```

When no DPP is assigned to a ROM memory range the linker will create a RAM copy of the near ROM data sections.

Example 1: Partial caching of a flash array

If you want to have only a part of a flash array cached, you should use `group/select` statements. In this example there are two sets of code sections: a set of sections that must be cached and a set of sections that shall not be cached. The section names are defined with `#pragma section` in the C code:

```
#pragma section code=cached_{name}
// ... functions that shall be located in cached memory ...

#pragma section code=not_cached_{name}
// ... functions that shall be located in not cached memory ...
```

In the LSL file the following section layout can be defined:

```
section_layout ::code
{
    group( run_addr = mem:spe:Flash0/cached )
    {
        select "cached_*";
    }
    group( run_addr = mem:spe:Flash0/not_cached )
    {
        select "not_cached_*";
    }
}
```

In this example the linker is free to choose where to put the sections in the full `Flash0` range. The locations of sections that are located in the `cached` range are also occupied in the `not_cached` mirror and vice versa.

Instead of specifying the memory range with the `mem:` reference, you can also specify fixed ranges. This also allows you to separate the memory ranges strictly:

```
section_layout ::code
{
    group( run_addr = [0x800000..0x81FFFF] )
    {
        select "cached_*";
    }
    group( run_addr = [0xC20000..0xC3FFFF] )
```

```
    {
        select "not_cached_*";
    }
}
```

Keep in mind that once a section is selected, it will not be selected again with a subsequent `select` statement. For example, if you want section "`cached_foo`" to be located at an absolute address, say 0x810000, this must be done before the wild-card selection:

```
section_layout ::code
{
    group( run_addr = 0x810000 )
    {
        select "cached_foo";
    }
    group( run_addr = [0x800000..0x81FFFF] )
    {
        select "cached_*";
    }
    group( run_addr = [0xC20000..0xC3FFFF] )
    {
        select "not_cached_*";
    }
}
```

Example 2: Mixing cached code and data in the same flash area

In this example the DPP address mappings are set as follows:

- `DPP0 => 0xC00000`

- `DPP1 => 0xC04000`

- `DPP2 => 0x008000`

- `DPP3 => 0x00C000`

This means that the area of 0xC000000..0xC07FFF must be used for near romdata.

In the memory definition, the mappings for `Flash0` are set as follows:

- `not_cached` memory at address 0xC00000 is reserved

- `cached` memory at address 0x800000 is *not* reserved

This effectively implies romdata addressed with DPP0 and DPP1 cannot be located freely because this address range is reserved (the `not_cached` memory range is reserved). To locate near romdata in this address range you must locate it explicitly by using an absolute address or address range:

```
section_layout ::near  // selections apply to near sections only
{
    group( run_addr = [0xC00000..0xC07FFF] )
```

```
    {
            // select all near ROM sections
        select (attributes=+r-w);
    }
}
```

A similar example can be made for far, shuge and huge sections.

An alternative method is to make both the `cached` memory mappings and the `not_cached` memory mappings not reserved. The linker is then free to locate code sections either in the cached or the not cached area. The linker usually uses the lowest address first, but may be directed otherwise by LSL statements. Therefore, this method requires that you inspect the generated map file carefully.

8.7.12. Duplicate Section Removal and Mirrors

When the linker detects that sections have equal contents it removes all duplicates and it lets all references point to the remaining section. When explicitly locating sections into mirrored ranges, it may happen that a section has a duplicate located in one of the mirrors. The linker removes one of the duplicates, with the result that the section is only located in one of the ranges: the original range or a mirror.

Example:

Say there are two identical functions `foo()` and `bar()`, defined as follows:

```
#pragma section code=cached_{name}
int foo( int a )
{
    return( 10*a );
}

#pragma section code=not_cached_{name}
int bar( int a )
{
    return( 10*a );
}
```

Here `foo()` and `bar()` are clearly the same.

The LSL defined for locating the generated sections looks as follows:

```
section_layout ::code
{
    group( run_addr = mem:spe:Flash0/cached )
    {
        select "cached_*";
    }
    group( run_addr = mem:spe:Flash0/not_cached )
    {
        select "not_cached_*";
    }
}
```

The linker detects that the sections generated for `foo()` and for `bar()` are identical and decides to remove the section (`not_cached_bar`) generated for the function `bar()`. The references, i.e., function calls to `bar()` now point to `foo()`, which is valid because it is identical. However, we intended to put `bar()` into not cached memory, while `foo()` is in cached memory. Effectively `bar()` now also resides in cached memory.

You can avoid this linker optimization by using the `#pragma protect` in the C code:

```
#pragma section code=cached_{name}
int foo( int a )
{
    return( 10*a );
}

#pragma section code=not_cached_{name}
#pragma protect
int bar( int a )
{
    return( 10*a );
}
#pragma protect restore
```

8.8. Linker Labels

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

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

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.
`__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.

> At C level, all linker labels start with one leading underscore (the compiler adds an extra underscore).

Additionally, the linker script file defines the following symbols:

Symbol	Description
`__lc_base_dpp0`	Alias for `__DPP0_ADDR`.
`__lc_base_dpp1`	Alias for `__DPP1_ADDR`.
`__lc_base_dpp2`	Alias for `__DPP2_ADDR`.
`__lc_base_dpp3`	Alias for `__DPP3_ADDR`.
`__lc_copy_table`	Start of copy table. Same as `__lc_ub_table`. The copy table gives the source and destination addresses of sections to be copied. This table will be generated by the linker only if this label is used.
`__lc_vector_table`	Start of vector table. Same as `__lc_vb_vector_table_0`.

Example: refer to the stack

Suppose in an LSL file a stack section is defined with the name "`user_stack`" (with the keyword `stack`). You can refer to the begin and end of the stack from your C source as follows (labels have one leading underscore):

```
#include <stdio.h>
extern char _lc_ub_user_stack[];
extern char _lc_ue_user_stack[];
void main()
{
  printf( "Size of stack is %d\n",
          _lc_ub_user_stack - _lc_ue_user_stack );
          /* stack grows from high to low */
}
```

From assembly you can refer to the end of the stack with:

```
.extern __lc_ue_user_stack   ; end of user stack
```

8.9. Generating a Map File

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

To generate a map file

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

    The Properties dialog appears.

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

In the right pane the Settings appear.

3. On the Tool Settings tab, select **Linker » Map File**.

4. Enable the option **Generate XML map file format (.mapxml) for map file viewer**.

5. (Optional) Enable the option **Generate map file (.map)**.

6. (Optional) Enable the options to include that information in the map file.

Example on the command line (Windows Command Prompt)

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

```
lk166 --map-file test.obj
```

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

See Section 14.2, *Linker Map File Format*, for an explanation of the format of the map file.

8.10. Linker Error Messages

The linker reports the following types of error messages in the Problems view of Eclipse.

F (Fatal errors)

After a fatal error the linker immediately aborts the link/locate process.

E (Errors)

Errors are reported, but the linker continues linking and locating. No output files are produced unless you have set the linker option**--keep-output-files**.

W (Warnings)

Warning messages do not result into an erroneous output file. They are meant to draw your attention to assumptions of the linker for a situation which may not be correct. You can control warnings in the **C/C++ Build » Settings » Tool Settings » Linker » Diagnostics** page of the **Project » Properties for** menu (linker option **--no-warnings**).

I (Information)

Verbose information messages do not indicate an error but tell something about a process or the state of the linker. To see verbose information, use the linker option**--verbose**.

S (System errors)

System errors occur when internal consistency checks fail and should never occur. When you still receive the system error message

```
S6##: message
```

please report the error number and as many details as possible about the context in which the error occurred.

Display detailed information on diagnostics

1.   From the **Window** menu, select **Show View » Other » General » Problems**.

      The Problems view is added to the current perspective.

2.   In the Problems view right-click on a message.

      A popup menu appears.

3.   Select **Detailed Diagnostics Info**.

      A dialog box appears with additional information.

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

```
lk166 --diag=[format:]{all | number,...]
```

Chapter 9. Using the Utilities

The TASKING VX-toolset for C166 comes with a number of utilities:

cc166   A control program. The control program invokes all tools in the toolset and lets you quickly generate an absolute object file from C and/or assembly source input files. Eclipse uses the control program to call the compiler, assembler and linker.

mk166   A utility program to maintain, update, and reconstruct groups of programs. The make utility looks whether files are out of date, rebuilds them and determines which other files as a consequence also need to be rebuilt.

amk   The make utility which is used in Eclipse. It supports parallelism which utilizes the multiple cores found on modern host hardware.

ar166   An archiver. With this utility you create and maintain library files with relocatable object modules (`.obj`) generated by the assembler.

9.1. Control Program

The control program is a tool that invokes all tools in the toolset for you. It provides a quick and easy way to generate the final absolute object file out of your C/C++ sources without the need to invoke the compiler, assembler and linker manually.

Eclipse uses the control program to call the C++ compiler, C compiler, assembler and linker, but you can call the control program from the command line. The invocation syntax is:

cc166 [[*option*]... [*file*]...]...

Recognized input files

* Files with a `.cc`, `.cxx` or `.cpp` suffix are interpreted as C++ source programs and are passed to the C++ compiler.

* Arguments with a `.c` suffix are interpreted as C source programs and are passed to the compiler.

* Files with a `.asm` suffix are interpreted as hand-written assembly source files which have to be passed to the assembler.

* Files with a `.src` suffix are interpreted as compiled assembly source files. They are directly passed to the assembler.

* Files with a `.lib` suffix are interpreted as library files and are passed to the linker.

* Files with a `.obj` suffix are interpreted as object files and are passed to the linker.

* Files with a `.out` suffix are interpreted as linked object files and are passed to the locating phase of the linker. The linker accepts only one `.out` file in the invocation.

* Files with a `.lsl` suffix are interpreted as linker script files and are passed to the linker.

Options

The control program accepts several command line options. If you specify an unknown option to the control program, the control program looks if it is an option for a specific tool. If so, it passes the option directly to the tool. However, it is recommended to use the control program options **--pass-*** (**-Wcp**, **-Wc**, **-Wa**, **-Wl**) to pass arguments directly to tools.

For a complete list and description of all control program options, see Section 12.6, *Control Program Options*.

Example with verbose output

```
cc166 --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\c166" -Mn -o cc3248a.src test.c
+ "path\as166" -o cc3248b.obj cc3248a.src
+ "path\lk166" -o test.elf -D__CPU__=c16x --map-file
    cc3248b.obj -lcn -lfpn -lrtn "-Lpath\lib\l"
```

The control program produces unique filenames for intermediate steps in the compilation process (such as `cc3248a.src` and `cc3248b.obj` 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

```
cc166 --pass-compiler=-Oc test.c
```

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

9.2. Make Utility mk166

If you are working with large quantities of files, or if you need to build several targets, it is rather time-consuming to call the individual tools to compile, assemble, link and locate all your files.

You save already a lot of typing if you use the control program and define an options file. You can even create a batch file or script that invokes the control program for each target you want to create. But with these methods all files are completely compiled, assembled and linked to obtain the target file, even if you changed just one C source. This may demand a lot of (CPU) time on your host.

The make utility **mk166** is a tool to maintain, update, and reconstruct groups of programs. The make utility looks which files are out-of-date and only recreates these files to obtain the updated target.

Make process

In order to build a target, the make utility needs the following input:

• the target it should build, specified as argument on the command line

• the rules to build the target, stored in a file usually called makefile

> In addition, the make utility also reads the file mk166.mk which contains predefined rules and macros. See Section 9.2.2, *Writing a Makefile*.

The makefile contains the relationships among your files (called *dependencies*) and the commands that are necessary to create each of the files (called *rules*). Typically, the absolute object file (.elf) is updated when one of its dependencies has changed. The absolute file depends on .obj files and libraries that must be linked together. The .obj files on their turn depend on .src files that must be assembled and finally, .src files depend on the C source files (.c) that must be compiled. In the makefile this looks like:

```
test.src : test.c                    # dependency
          c166 test.c                # rule

test.obj : test.src
          as166 test.src

test.elf : test.obj
          lk166 test.obj -o test.elf --map-file -lcn -lfpn -lrtn
```

You can use any command that is valid on the command line as a rule in the makefile. So, rules are not restricted to invocation of the toolset.

Example

To build the target test.elf, call **mk166** with one of the following lines:

```
mk166 test.elf

mk166 -fmymake.mak test.elf
```

By default the make utility reads the file `makefile` so you do not need to specify it on the command line. If you want to use another name for the makefile, use the option **-f**.

If you do not specify a target, **mk166** uses the first target defined in the makefile. In this example it would build `test.src` instead of `test.elf`.

Based on the sample invocation, the make utility now tries to build `test.elf` based on the makefile and performs the following steps:

1. From the makefile the make utility reads that `test.elf` depends on `test.obj`.

2. If `test.obj` does not exist or is out-of-date, the make utility first tries to build this file and reads from the makefile that `test.obj` depends on `test.src`.

3. If `test.src` does exist, the make utility now creates `test.obj` by executing the rule for it: `as166 test.src`.

4. There are no other files necessary to create `test.elf` so the make utility now can use `test.obj` to create `test.elf` by executing the rule: `lk166 test.obj -o test.elf ...`

The make utility has now built `test.elf` but it only used the assembler to update `test.obj` and the linker to create `test.elf`.

If you compare this to the control program:

```
cc166 test.c
```

This invocation has the same effect but now *all files* are recompiled (assembled, linked and located).

9.2.1. Calling the Make Utility

You can only call the make utility from the command line. The invocation syntax is:

mk166 [[*option*]... [*target*]... [*macro=def*]...]

For example:

```
mk166 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 **mk166** invocation. It overrides any regular definitions for the specified macro within the makefiles and from the environment. It is inherited by subordinate **mk166**'s but act as an environment variable for these. That is, depending on the **-e** setting, it may be overridden by a makefile definition.
option	For a complete list and description of all make utility options, see Section 12.7, *Make Utility Options*.

Exit status

The make utility returns an exit status of 1 when it halts as a result of an error. Otherwise it returns an exit status of 0.

9.2.2. Writing a Makefile

In addition to the standard makefile `makefile`, the make utility always reads the makefile `mk166.mk` before other inputs. This system makefile contains implicit rules and predefined macros that you can use in the makefile `makefile`.

With the option **-r** (Do not read the `mk166.mk` file) you can prevent the make utility from reading `mk166.mk`.

The default name of the makefile is `makefile` in the current directory. If you want to use another makefile, use the option **-f**.

The makefile can contain a mixture of:

- targets and dependencies

- rules

- macro definitions or functions

- conditional processing

- comment lines

- include lines

- export lines

To continue a line on the next line, terminate it with a backslash (****):

```
# this comment line is continued\
on the next line
```

If a line must end with a backslash, add an empty macro:

```
# this comment line ends with a backslash \$(EMPTY)
# this is a new line
```

9.2.2.1. Targets and Dependencies

The basis of the makefile is a set of targets, dependencies and rules. A target entry in the makefile has the following format:

```
target ... : [dependency ...] [; rule]
       [rule]
        ...
```

Target lines must always start at the beginning of a line, leading white spaces (tabs or spaces) are not allowed. A target line consists of one or more targets, a semicolon and a set of files which are required to build the target (*dependencies*). The target itself can be one or more filenames or symbolic names:

```
all:                    demo.elf final.elf

demo.elf final.elf:     test.obj demo.obj final.obj
```

You can now can specify the target you want to build to the make utility. The following three invocations all have the same effect:

```
mk166
mk166 all
mk166 demo.elf final.elf
```

If you do *not* specify a target, the first target in the makefile (in this example `all`) is built. The target `all` depends on `demo.elf` and `final.elf` so the second and third invocation have the same effect and the files `demo.elf` and `final.elf` are built.

You can normally use colons to denote drive letters. The following works as intended:

```
c:foo.obj : a:foo.c
```

If a target is defined in more than one target line, the dependencies are added to form the target's complete dependency list:

```
all:  demo.elf   # These two lines are equivalent with:
all:  final.elf  # all: demo.elf final.elf
```

Special targets

There are a number of special targets. Their names begin with a period.

Target	Description
`.DEFAULT`	If you call the make utility with a target that has no definition in the makefile, this target is built.
`.DONE`	When the make utility has finished building the specified targets, it continues with the rules following this target.
`.IGNORE`	Non-zero error codes returned from commands are ignored. Encountering this in a makefile is the same as specifying the option **-i** on the command line.
`.INIT`	The rules following this target are executed before any other targets are built.
`.PRECIOUS`	Dependency files mentioned for this target are never removed. Normally, if a command in a rule returns an error or when the target construction is interrupted, the make utility removes that target file. You can use the option **-p** on the command line to make all targets precious.
`.SILENT`	Commands are not echoed before executing them. Encountering this in a makefile is the same as specifying the option **-s** on the command line.

Target	Description
`.SUFFIXES`	This target specifies a list of file extensions. Instead of building a completely specified target, you now can build a target that has a certain file extension. Implicit rules to build files with a number of extensions are included in the system makefile `mk166.mk`. If you specify this target with dependencies, these are added to the existing `.SUFFIXES` target in `mk166.mk`. If you specify this target without dependencies, the existing list is cleared.

9.2.2.2. Makefile Rules

A line with leading white space (tabs or spaces) is considered as a rule and associated with the most recently preceding dependency line. A *rule* is a line with commands that are executed to build the associated target. A target-dependency line can be followed by one or more rules.

```
final.src : final.c                # target and dependency
            move test.c final.c    # rule1
            c166 final.c           # rule2
```

You can precede a rule with one or more of the following characters:

@   does not echo the command line, except if **-n** is used.

-          the make utility ignores the exit code of the command. Normally the make utility stops if a non-zero exit code is returned. This is the same as specifying the option **-i** on the command line or specifying the special `.IGNORE` target.

+   The make utility uses a shell or Windows command prompt (`cmd.exe`) to execute the command. If the '**+**' is not followed by a shell line, but the command is an MS-DOS command or if redirection is used (<, |, >), the shell line is passed to `cmd.exe` anyway.

You can force **mk166** to execute multiple command lines in one shell environment. This is accomplished with the token combination '**;**'. For example:

```
cd c:\Tasking\bin ;\
mk166 -V
```

Note that the ';' must always directly be followed by the '\' token. Whitespace is not removed when it is at the end of the previous command line or when it is in front of the next command line. The use of the ';' as an operator for a command (like a semicolon ';' separated list with each item on one line) and the '\' as a layout tool is not supported, unless they are separated with whitespace.

Inline temporary files

The make utility can generate inline temporary files. If a line contains **<<***LABEL* (no whitespaces!) then all subsequent lines are placed in a temporary file until the line *LABEL* is encountered. Next, **<<***LABEL* is replaced by the name of the temporary file. For example:

```
lk166 -o $@ -f <<EOF
    $(separate "\n" $(match .obj $!))
    $(separate "\n" $(match .lib $!))
```

```
      $(LKFLAGS)
EOF
```

The three lines between `<<EOF` and `EOF` are written to a temporary file (for example `mkce4c0a.tmp`), and the rule is rewritten as: `lk166 -o $@ -f mkce4c0a.tmp`.

Suffix targets

Instead of specifying a specific target, you can also define a general target. A general target specifies the rules to generate a file with extension `.ex1` to a file with extension `.ex2`. For example:

```
.SUFFIXES:   .c
.c.obj   :
             cc166 -c $<
```

Read this as: to build a file with extension `.obj` out of a file with extension `.c`, call the control program with **-c $<**. **$<** is a predefined macro that is replaced with the name of the current dependency file. The special target `.SUFFIXES`: is followed by a list of file extensions of the files that are required to build the target.

Implicit rules

Implicit rules are stored in the system makefile `mk166.mk` and are intimately tied to the `.SUFFIXES` special target. Each dependency that follows the `.SUFFIXES` target, defines an extension to a filename which must be used to build another file. The implicit rules then define how to actually build one file from another. These files share a common basename, but have different extensions.

If the specified target on the command line is not defined in the makefile or has not rules in the makefile, the make utility looks if there is an implicit rule to build the target.

Example:

```
LIB  =     -lcn -lfpn -lrtn    # macro

prog.elf:  prog.obj sub.obj
     lk166 prog.obj sub.obj $(LIB) -o prog.elf

prog.obj:  prog.c inc.h
     c166  prog.c
     as166 prog.src

sub.obj:   sub.c inc.h
     c166  sub.c
     as166 sub.src
```

This makefile says that `prog.elf` depends on two files `prog.obj` and `sub.obj`, and that they in turn depend on their corresponding source files (`prog.c` and `sub.c`) along with the common file `inc.h`.

The following makefile uses implicit rules (from `mk166.mk`) to perform the same job.

```
LDFLAGS = -lcn -lfpn -lrtn              # macro used by implicit rules
prog.elf: prog.obj sub.obj              # implicit rule used
```

```
prog.obj: prog.c inc.h                  # implicit rule used
sub.obj:  sub.c inc.h                   # implicit rule used
```

9.2.2.3. Macro Definitions

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

```
MACRO = text
MACRO += and more text
```

Spaces around the equal sign are not significant. With the **+=** operator you can add a string to an existing macro. An extra space is inserted before the added string automatically.

To use a macro, you must access its contents:

```
$(MACRO)         # you can read this as
${MACRO}         # the contents of macro MACRO
```

If the macro name is a single character, the parentheses are optional. Note that the expansion is done recursively, so the body of a macro may contain other macros. These macros are expanded when the macro is actually used, not at the point of definition:

```
FOOD = $(EAT) and $(DRINK)
EAT = meat and/or vegetables
DRINK = water
export FOOD
```

The macro FOOD is expanded as meat and/or vegetables and water at the moment it is used in the export line, and the environment variable FOOD is set accordingly.

Predefined macros

Macro	Description
MAKE	Holds the value **mk166**. Any line which uses MAKE, temporarily overrides the option **-n** (Show commands without executing), just for the duration of the one line. This way you can test nested calls to MAKE with the option **-n**.
MAKEFLAGS	Holds the set of options provided to **mk166** (except for the options **-f** and **-d**). If this macro is exported to set the environment variable MAKEFLAGS, the set of options is processed before any command line options. You can pass this macro explicitly to nested **mk166**'s, but it is also available to these invocations as an environment variable.

PRODDIR	Holds the name of the directory where **mk166** is installed. You can use this macro to refer to files belonging to the product, for example a library source file.
	
	`DOPRINT = $(PRODDIR)/lib/src/_doprint.c`
	
	When **mk166** is installed in the directory c:/Tasking/bin this line expands to:
	
	`DOPRINT = c:/Tasking/lib/src/_doprint.c`

Macro	Description
`SHELLCMD`	Holds the default list of commands which are local to the SHELL. If a rule is an invocation of one of these commands, a SHELL is automatically spawned to handle it.
`$`	This macro translates to a dollar sign. Thus you can use "$$" in the makefile to represent a single "$".

Dynamically maintained macros

There are several dynamically maintained macros that are useful as abbreviations within rules. It is best not to define them explicitly.

Macro	Description
`$*`	The basename of the current target.
`$<`	The name of the current dependency file.
`$@`	The name of the current target.
`$?`	The names of dependents which are younger than the target.
`$!`	The names of all dependents.

The $< and $* macros are normally used for implicit rules. They may be unreliable when used within explicit target command lines. All macros may be suffixed with **F** to specify the Filename components (e.g. `${*F}`, `${@F}`). Likewise, the macros $*, $< and $@ may be suffixed by **D** to specify the Directory component.

The result of the $* macro is always without double quotes ("), regardless of the original target having double quotes (") around it or not.

The result of using the suffix **F** (Filename component) or **D** (Directory component) is also always without double quotes ("), regardless of the original contents having double quotes (") around it or not.

9.2.2.4. Makefile Functions

A function not only expands but also performs a certain operation. Functions syntactically look like macros but have embedded spaces in the macro name, e.g. '$(match arg1 arg2 arg3)'. All functions are built-in and currently there are five of them: `match`, `separate`, `protect`, `exist` and `nexist`.

match

The `match` function yields all arguments which match a certain suffix:

```
$(match .obj prog.obj sub.obj mylib.lib)
```

yields:

```
prog.obj sub.obj
```

separate

The separate function concatenates its arguments using the first argument as the separator. If the first argument is enclosed in double quotes then '\n' is interpreted as a newline character, '\t' is interpreted as a tab, '*ooo*' is interpreted as an octal value (where, *ooo* is one to three octal digits), and spaces are taken literally. For example:

```
$(separate "\n" prog.obj sub.obj)
```

results in:

```
prog.obj
sub.obj
```

Function arguments may be macros or functions themselves. So,

```
$(separate "\n" $(match .obj $!))
```

yields all object files the current target depends on, separated by a newline string.

protect

The protect function adds one level of quoting. This function has one argument which can contain white space. If the argument contains any white space, single quotes, double quotes, or backslashes, it is enclosed in double quotes. In addition, any double quote or backslash is escaped with a backslash.

Example:

```
echo $(protect I'll show you the "protect" function)
```

yields:

```
echo "I'll show you the \"protect\" function"
```

exist

The exist function expands to its second argument if the first argument is an existing file or directory.

Example:

```
$(exist test.c cc166 test.c)
```

When the file test.c exists, it yields:

```
cc166 test.c
```

When the file test.c does not exist nothing is expanded.

nexist

The `nexist` function is the opposite of the `exist` function. It expands to its second argument if the first argument is not an existing file or directory.

Example:

```
$(nexist test.src cc166 test.c)
```

9.2.2.5. Conditional Processing

Lines containing `ifdef`, `ifndef`, `else` or `endif` are used for conditional processing of the makefile. They are used in the following way:

```
ifdef macro-name
if-lines
else
else-lines
endif
```

The *if-lines* and *else-lines* may contain any number of lines or text of any kind, even other `ifdef`, `ifndef`, `else` and `endif` lines, or no lines at all. The `else` line may be omitted, along with the *else-lines* following it.

First the *macro-name* after the `ifdef` command is checked for definition. If the macro is defined then the *if-lines* are interpreted and the *else-lines* are discarded (if present). Otherwise the *if-lines* are discarded; and if there is an `else` line, the *else-lines* are interpreted; but if there is no else line, then no lines are interpreted.

When you use the `ifndef` line instead of `ifdef`, the macro is tested for not being defined. These conditional lines can be nested up to 6 levels deep.

You can also add tests based on strings. With `ifeq` the result is true if the two strings match, with `ifneq` the result is true if the two strings do not match. They are used in the following way:

```
ifeq(string1,string2)
if-lines
else
else-lines
endif
```

9.2.2.6. Comment, Include and Export Lines

Comment lines

Anything after a "#" is considered as a comment, and is ignored. If the "#" is inside a quoted string, it is not treated as a comment. Completely blank lines are ignored.

```
test.src  : test.c        # this is comment and is
            cc166 test.c  # ignored by the make utility
```

Include lines

An *include line* is used to include the text of another makefile (like including a `.h` file in a C source). Macros in the name of the included file are expanded before the file is included. You can include several files. Include files may be nested.

```
include makefile2 makefile3
```

Export lines

An *export line* is used to export a macro definition to the environment of any command executed by the make utility.

```
GREETING = Hello
export GREETING
```

This example creates the environment variable `GREETING` with the value `Hello`. The macro is exported at the moment the export line is read so the macro definition has to precede the export line.

9.3. Make Utility amk

amk is the make utility Eclipse uses to maintain, update, and reconstruct groups of programs. But you can also use it on the command line. Its features are a little different from **mk166**. The main difference compared to **mk166** and other make utilities, is that **amk** features parallelism which utilizes the multiple cores found on modern host hardware, hardening for path names with embedded white space and it has an (internal) interface to provide progress information for updating a progress bar. It does not use an external command shell (/bin/sh, cmd.exe) but executes commands directly.

The primary purpose of any make utility is to speed up the edit-build-test cycle. To avoid having to build everything from scratch even when only one source file changes, it is necessary to describe dependencies between source files and output files and the commands needed for updating the output files. This is done in a so called "makefile".

9.3.1. Makefile Rules

A makefile dependency rule is a single line of the form:

```
[target ...] : [prerequisite ...]
```

where *target* and *prerequisite* are path names to files. Example:

```
test.obj : test.c
```

This states that target `test.obj` depends on prerequisite `test.c`. So, whenever the latter is modified the first must be updated. Dependencies accumulate: prerequisites and targets can be mentioned in multiple dependency rules (circular dependencies are not allowed however). The command(s) for updating a target when any of its prerequisites have been modified must be specified with leading white space after any of the dependency rule(s) for the target in question. Example:

```
test.obj :
  cc166 test.c   # leading white space
```

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

```
test.obj : test.c
  cc166 test.c
```

White space around the colon is not required. When a path name contains special characters such as '**:**', '**#**' (start of comment), '**=**' (macro assignment) or any white space, then the path name must be enclosed in single or double quotes. Quoted strings can contain anything except the quote character itself and a newline. Two strings without white space in between are interpreted as one, so it is possible to embed single and double quotes themselves by switching the quote character.

When a target does not exist, its modification time is assumed to be very old. So, **amk** will try to make it. When a prerequisite does not exist possibly after having tried to make it, it is assumed to be very new. So, the update commands for the current target will be executed in that case. **amk** will only try to make targets which are specified on the command line. The default target is the first target in the makefile which does not start with a dot.

Static pattern rules

Static pattern rules are rules which specify multiple targets and construct the prerequisite names for each target based on the target name.

```
[target ...] : target-pattern : [prerequisite-patterns ...]
```

The *target* specifies the targets the rules applies to. The *target-pattern* and *prerequisite-patterns* specify how to compute the prerequisites of each target. Each target is matched against the *target-pattern* to extract a part of the target name, called the *stem*. This stem is substituted into each of the *prerequisite-patterns* to make the prerequisite names (one from each *prerequisite-pattern*).

Each pattern normally contains the character '%' just once. When the *target-pattern* matches a target, the '%' can match any part of the target name; this part is called the *stem*. The rest of the pattern must match exactly. For example, the target `foo.obj` matches the pattern '`%.obj`', with '`foo`' as the stem. The targets `foo.c` and `foo.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.obj filter.obj

all: $(objects)

$(objects): %.obj: %.c
    cc166 -c $< -o $@
    echo the stem is $*
```

Here '`$<`' is the automatic variable that holds the name of the prerequisite, '`$@`' is the automatic variable that holds the name of the target and '`$*`' is the stem that matches the pattern. Internally this translates to the following two rules:

```
test.obj: test.c
    cc166 -c test.c -o test.obj
    echo the stem is test

filter.obj: filter.c
    cc166 -c filter.c -o filter.obj
    echo the stem is filter
```

Each target specified must match the target pattern; a warning is issued for each target that does not.

Special targets

There are a number of special targets. Their names begin with a period.

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 *.obj`
	
	With `amk clean`, the command is executed regardless of whether there is a file named `clean`.

9.3.2. Makefile Directives

Directives inside makefiles are executed while reading the makefile. When a line starts with the word "`include`" or "`-include`" then the remaining arguments on that line are considered filenames whose contents are to be inserted at the current line. "`-include`" will silently skip files which are not present. You can include several files. Include files may be nested.

Example:

```
include makefile2 makefile3
```

White spaces (tabs or spaces) in front of the directive are allowed.

9.3.3. 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:.obj"=.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.

9.3.4. 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.obj 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.obj readme.txt .project output.c}
```

results in:

```
test.obj 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
```

9.3.5. Conditional Processing

Lines containing `ifdef`, `ifndef`, `else` or `endif` are used for conditional processing of the makefile. They are used in the following way:

```
ifdef macro-name
if-lines
else
else-lines
endif
```

The *if-lines* and *else-lines* may contain any number of lines or text of any kind, even other `ifdef`, `ifndef`, `else` and `endif` lines, or no lines at all. The `else` line may be omitted, along with the *else-lines* following it. White spaces (tabs or spaces) in front of preprocessing directives are allowed.

First the *macro-name* after the `ifdef` command is checked for definition. If the macro is defined then the *if-lines* are interpreted and the *else-lines* are discarded (if present). Otherwise the *if-lines* are discarded; and if there is an `else` line, the *else-lines* are interpreted; but if there is no else line, then no lines are interpreted.

When you use the `ifndef` line instead of `ifdef`, the macro is tested for not being defined. These conditional lines can be nested to any level.

You can also add tests based on strings. With `ifeq` the result is true if the two strings match, with `ifneq` the result is true if the two strings do not match. They are used in the following way:

```
ifeq(string1,string2)
if-lines
else
else-lines
endif
```

9.3.6. Makefile Parsing

amk reads and interprets a makefile in the following order:

1. When the last character on a line is a backslash (****) (i.e. without trailing white space) then that line and the next line will be concatenated, removing the backslash and newline.

2. The unquoted '**#**' character indicates start of comment and may be placed anywhere on a line. It will be removed in this phase.

   ```
   # this comment line is continued\
   on the next line
   ```

3. Trailing white space is removed.

4. When a line starts with white space and it is not followed by a directive or preprocessing directive, then it is interpreted as a command for updating a target.

5. Otherwise, when a line contains the unquoted text '**=**', '**+=**' or '**:=**' operator, then it will be interpreted as a macro definition.

6. Otherwise, all macros on the line are evaluated before considering the next steps.

7. When the resulting line contains an unquoted '**:**' the line is interpreted as a dependency rule.

8. When the first token on the line is "`include`" or "`-include`" (which by now must start on the first column of the line), **amk** will execute the directive.

9. Otherwise, the line must be empty.

Macros in commands for updating a target are evaluated right before the actual execution takes place (or would take place when you use the **-n** option).

9.3.7. 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.

9.3.8. 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 12.8, *Parallel Make Utility Options*.

Exit status

The make utility returns an exit status of 1 when it halts as a result of an error. Otherwise it returns an exit status of 0.

9.4. Archiver

The archiver **ar166** is a program to build and maintain your own library files. A library file is a file with extension `.lib` and contains one or more object files (`.obj`) that may be used by the linker.

The archiver has five main functions:

* Deleting an object module from the library

* Moving an object module to another position in the library file

* Replacing an object module in the library or add a new object module

* Showing a table of contents of the library file

* Extracting an object module from the library

The archiver takes the following files for input and output:
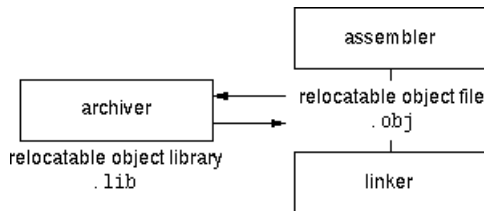
The linker optionally includes object modules from a library if that module resolves an external symbol definition in one of the modules that are read before.

9.4.1. Calling the Archiver

You can create a library in Eclipse, which calls the archiver or you can call the archiver on the command line.

To create a library in Eclipse

Instead of creating a C166 absolute ELF file, you can choose to create a library. You do this when you create a new project with the New C/C++ Project wizard.

1. From the **File** menu, select **New » TASKING VX-toolset for C166 C/C++ Project**.

   The New C/C++ Project wizard appears.

2. Enter a project name.

3. In the **Project type** box, select **TASKING C166 Library** and click **Next >**.

4. Follow the rest of the wizard and click **Finish**.

5. Add the files to your project.

6. Build the project as usual. For example, select **Project » Build Project** (📼).

   Eclipse builds the library. Instead of calling the linker, Eclipse now calls the archiver.

Command line invocation

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

ar166 *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		

Description	Option	Sub-option
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 12.9, *Archiver Options*.

9.4.2. Archiver Examples

Create a new library

If you add modules to a library that does not yet exist, the library is created. To create a new library with the name `mylib.lib` and add the object modules `cstart.obj` and `calc.obj` to it:

```
ar166 -r mylib.lib cstart.obj calc.obj
```

Add a new module to an existing library

If you add a new module to an existing library, the module is added at the end of the module. (If the module already exists in the library, it is replaced.)

```
ar166 -r mylib.lib mod3.obj
```

Print a list of object modules in the library

To inspect the contents of the library:

```
ar166 -t mylib.lib
```

The library has the following contents:

```
cstart.obj
calc.obj
mod3.obj
```

Move an object module to another position

To move `mod3.obj` to the beginning of the library, position it just before `cstart.obj`:

```
ar166 -mb cstart.obj mylib.lib mod3.obj
```

Delete an object module from the library

To delete the object module `cstart.obj` from the library `mylib.lib`:

```
ar166 -d mylib.lib cstart.obj
```

Extract all modules from the library

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

```
ar166 -x mylib.lib
```

9.5. HLL Object Dumper

The high level language (HLL) dumper **hldump166** is a program to dump information about an object file. Key features are a disassembler with HLL source intermixing and HLL symbol display and a HLL symbol listing of static and global symbols.

9.5.1. Invocation

Command line invocation

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

hldump166 [*option*]... *file*...

The input file must be an ELF file with or without DWARF debug info (`.obj`, `.lib`, `.out` or `.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:

```
========== objectfile.obj ==========
```

For a complete list and description of all options, see Section 12.10, *HLL Object Dumper Options*. With `hldump166 --help` you will see the options on `stdout`.

9.5.2. HLL Dump Output Format

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

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

1. Module list

2. Section list

3. Section dump (disassembly)

4. HLL symbol table

5. Assembly level symbol table

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:

```
cc166 -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 files `hello.obj` (the object file) and `hello.elf` (the absolute output file).

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

```
hldump166 hello.obj

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

Name     Full path
hello.c hello.c

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

Address   Size   Align Name
00010000      8      2 code_main
00000000     14      1 near__1_str

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

    .section data, 'near__1_str'
    .org 00000000
    .db 48,65,6C,6C,6F,20,77,6F,72,6C,64,21,0A,00      ; Hello world!..
    ; End of section


                                  .section code, 'code_main'
00010000 0000F2E6                 mov     r2, #0
00010004 000000FA                 jmps    0x0
                                  ; End of section

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

Address     Size HLL Type             Name

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

Address   Size    Type Name
00000000
00000000           main
00000000           hello.c
00000000      data _$1$str
00000000           near__1_str
00000000           code_main
00000000           ___printf_noarg
00000000           __cstart
00000000           _printf
00000000      code _main
00000000           ___libc_references
```

You can dump information about the absolute object file with the following command:

```
hldump166 hello.elf
```

Module list

This part lists all modules (C/C++ files) found in the object file(s). It lists the filename and the complete path name at the time the module was built.

Section list

This part lists all sections found in the object file(s).

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, .dw, .dl 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 5 (near__1_str):
00000200 48 65 6C 6C 6F 20 77 6F 72 6C 64 21 0A 00        Hello world!..
```

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:

```
0000F698    80 static char   __near stdin_buf[80] [_iob.c]
```

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

```
0000F738    10 struct        __near _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:

```
0000F738    10 struct        __near _dbg_request [dbg.c]
0000F738    2    int                 _errno
0000F73A    1    enum                nr
0000F73C    6    union               u
0000F73C    2      struct              exit
```

```
0000F73C      2       int                   status
0000F73C      4      struct            open
0000F73C      2        const char        * pathname
0000F73E      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:

```
0040044A    34 int     __huge __nousm fputc(int __near c, struct _iobuf * __near fp)
```

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:

```
0000F700      4 int                   count [file.c, somefunc()]
0000F704      4 int                   count [x\a.c]
0000F708      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 10. Using the Debugger

This chapter describes the debugger and how you can run and debug a C or C++ application. This chapter only describes the TASKING specific parts.

10.1. Reading the Eclipse Documentation

Before you start with this chapter, it is recommended to read the Eclipse documentation first. It provides general information about the debugging process. This chapter guides you through a number of examples using the TASKING debugger with simulation as target.

You can find the Eclipse documentation as follows:

1. Start Eclipse.

2. From the **Help** menu, select **Help Contents**.

    The help screen overlays the Eclipse Workbench.

3. In the left pane, select **C/C++ Development User Guide**.

4. Open the **Getting Started** entry and select **Debugging projects**.

    This Eclipse tutorial provides an overview of the debugging process. Be aware that the Eclipse example does not use the TASKING tools and TASKING debugger.

10.2. Creating a Customized Debug Configuration

Before you can debug a project, you need a Debug launch configuration. Such a configuration, identified by a name, contains all information about the debug project: which debugger is used, which project is used, which binary debug file is used, which perspective is used, ... and so forth.

When you created your project, a default launch configuration for the TASKING simulator is available. At any time you can change this configuration. If you want to debug on a target board, you have to create a custom debug configuration for your target board.

> To debug or run a project, you need at least one opened and active project in your workbench.
> In this chapter, it is assumed that the `myproject` is opened and active in your workbench.

Customize your debug configuration

To change or create your own debug configuration follow the steps below.

1. From the **Debug** menu, select **Debug Configurations...**

    The Debug Configurations dialog appears.

2. In the left pane, select the configuration you want to change, for example, **TASKING C/C++ Debugger » myproject.simulator**.

Or: click the **New launch configuration** button (🗋) to add a new configuration.

The next dialog appears.

The dialog shows several tabs.

Main tab

On the **Main** tab, you can set the properties for the debug configuration such as a name for the configuration and the project and the application binary file which are used when you choose this configuration.
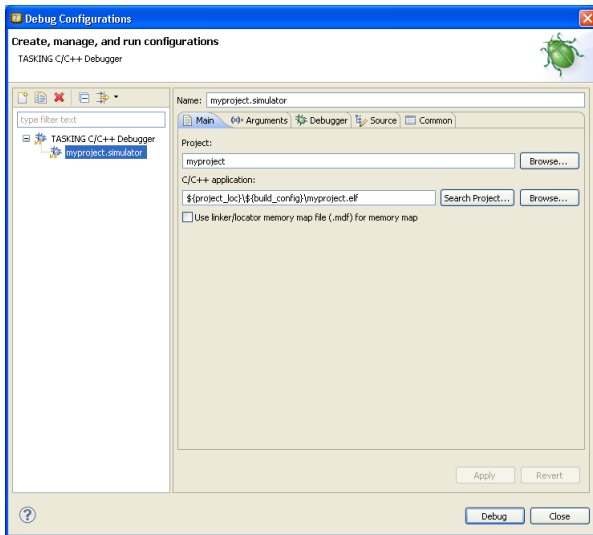
- **Name** is the name of the configuration. By default, this is the name of the project, optionally appended with `simulator` or `board`. You can give your configuration any name you want to distinguish it from the project name.

- In the **Project** field, you can choose the project for which you want to make a debug configuration. Because the project `myproject` is the active project, this project is filled in automatically. Click the Browse... button to select a different project. Only the *opened* projects in your workbench are listed.

- In the **C/C++ Application** field, you can choose the binary file to debug. The file `myproject.elf` is automatically selected from the active project.

- You can use the option **Use linker/locator memory map file (.mdf) for memory map** to find errors in your application that cause access to non-existent memory or cause an attempt to write to read-only memory. When building your project, the linker/locator creates a memory description file (`.mdf`) file which describes the memory regions of the target you selected in your project properties. The debugger uses this file to initialize the debugging target.

  This option is only useful in combination with a simulator as debug target. The debugger may fail to start if you use this option in combination with other debugging targets than a simulator.

Arguments tab

If your application's `main()` function takes arguments, you can pass them in this tab. Arguments are conventionally passed in the `argv[]` array. Because this array is allocated in target memory, make sure you have allocated sufficient memory space for it.

• In the C/C++ perspective select **Project » Properties for** to open the Properties dialog. Expand **C/C++ Build » Startup Configuration**. Enable the option **Enable passing argc/argv to main()** and specify a **Buffer size for argv**.
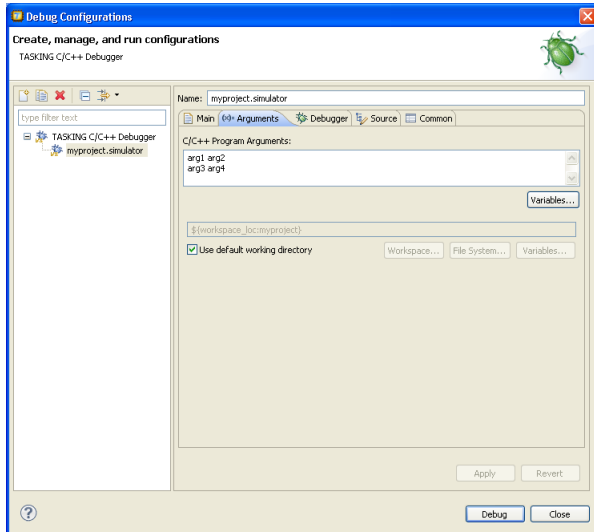
Debugger tab

On the **Debugger** tab you can set the debugger options. You can choose which debugger should be used and with what options it should work. The Debugger tab itself contains several tabs.
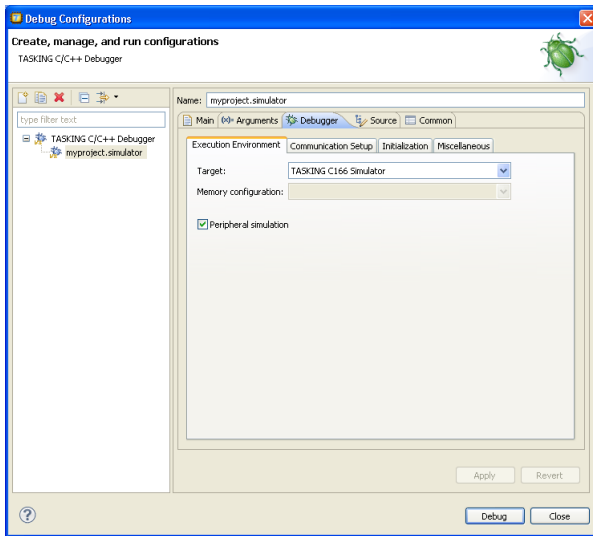
- On the **Execution Environment** tab you can select on which target the application should be debugged. An application can run on an external evaluation board, or on a simulator using your own PC. The information in this tab is based on the Debug Target Configuration (DTC) files as explained in Chapter 17, *Debug Target Configuration Files*.

- On the **Communication Setup** tab you can select the communication method and specific settings for the selected method (RS-232, TCP/IP, CAN) for execution environments. For the simulator this tab is grayed out.

- On the **Initialization** tab enable one or more of the following options:

  - **Initial download of program**

    If enabled, the target application is downloaded onto the target. If disabled, only the debug information in the file is loaded, which may be useful when the application has already been downloaded (or flashed) earlier. If downloading fails, the debugger will shut down.

  - **Verify download of program**

    If enabled, the debugger verifies whether the code and data has been downloaded successfully. This takes some extra time but may be useful if the connection to the target is unreliable.

  - **Program flash when downloading**

    If enabled, also flash devices are programmed (if necessary). Flash programming will not work when you use a simulator.

  - **Reset target**

    If enabled, the target is immediately reset after downloading has completed.

  - **Goto main**

If enabled, only the C startup code is processed when the debugger is launched. The application stops executing when it reaches the first C instruction in the function `main()`. Usually you enable this option in combination with the option **Reset Target**.

- **Break on exit**

If enabled, the target halts automatically when the `exit()` function is called.

- **Reduce target state polling**

If you have set a breakpoint, the debugger checks the status of the target every *number* of seconds to find out if the breakpoint is hit. In this field you can change the polling frequency.

- **Monitor file (Flash settings)**

Filename of the monitor, usually an Intel Hex or S-Record file.

- **Sector buffer size (Flash settings)**

Specifies the buffer size for buffering a flash sector.

- **Workspace address (Flash settings)**

The address of the workspace of the flash programming monitor.

- On the **Miscellaneous** tab you can specify several file locations.

  - **Debugger location**

  The location of the debugger itself. This should not be changed.

  - **FSS root directory**

  The initial directory used by file system simulation (FSS) calls. See the description of the FSS view.

  - **ORTI file** and **KSM module**

  If you wish to use the debugger's special facilities for OSEK kernels, specify the name of your ORTI file and that of your KSM module (shared library) in the appropriate edit boxes. See also the description of the RTOS view.

  - **GDI log file** and **Debug instrument log file**

  You can use the options GDI log file and Debug instrument log file (if applicable) to control the generation of internal log files. These are primarily intended for use by or at the request of Altium support personnel.

  - **Cache target access**

  Except when using a simulator, the debugger's performance is generally strongly dependent on the throughput and latency of the connection to the target. Depending on the situation, enabling this option may result in a noticeable improvement, as the debugger will then avoid re-reading registers and memory while the target remains halted. However, be aware that this may cause the debugger

to show the wrong data if tasks with a higher priority or external sources can influence the halted target's state.

Source tab

On the **Source** tab, you can add additional source code locations in which the debugger should search for debug data.
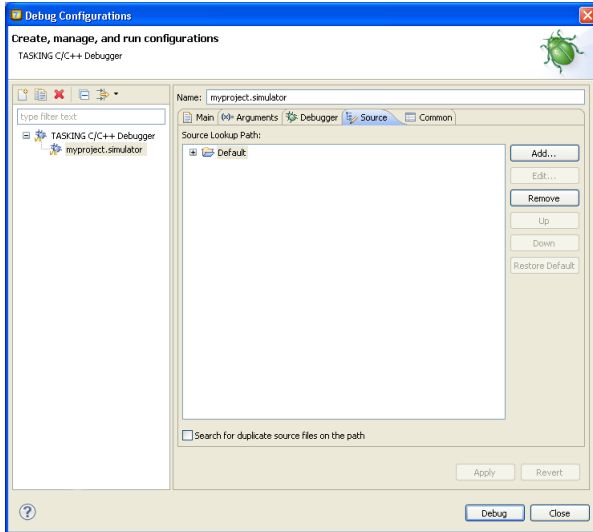
• Usually, the default source code location is correct.

Common tab

On the **Common** tab you can set additional launch configuration settings.
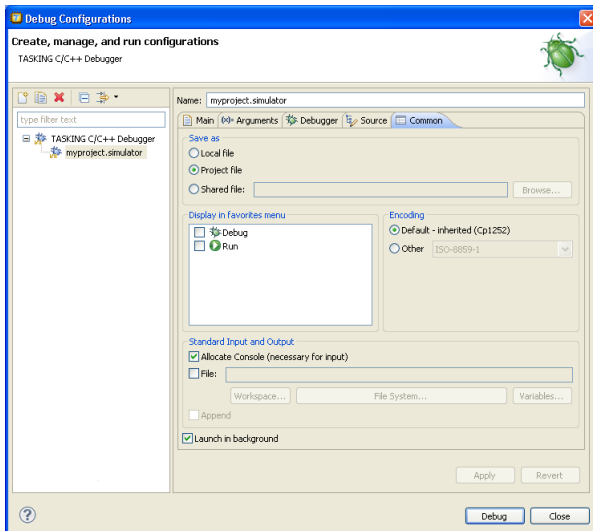
10.3. Troubleshooting

If the debugger does not launch properly, this is likely due to mistakes in the settings of the execution environment or to an improper connection between the host computer and the execution environment. Always read the notes for your particular execution environment.

Some common problems you may check for, are:

Problem	Solution
Wrong device name in the launch configuration	Make sure the specified device name is correct.
Invalid baud rate	Specify baud rate that matches the baud rate the execution environment is configured to expect.
No power to the execution environment.	Make sure the execution environment or attached probe is powered.
Wrong type of RS–232 cable.	Make sure you are using the correct type of RS-232 cable.
Cable connected to the wrong port on the execution environment or host.	Some target machines and hosts have several ports. Make sure you connect the cable to the correct port.
Conflict between communication ports.	A device driver or background application may use the same communications port on the host system as the debugger. Disable any service that uses the same port-number or choose a different port-number if possible.
Port already in use by another user.	The port may already be in use by another user on some UNIX hosts, or being allocated by a login process. Some target machines and hosts have several ports. Make sure you connect the cable to the correct port.

10.4. TASKING Debug Perspective

After you have launched the debugger, you are either asked if the TASKING Debug perspective should be opened or it is opened automatically. The Debug perspective consists of several views.

To open views in the Debug perspective:

1. Make sure the Debug perspective is opened

2. From the **Window** menu, select **Show View »**

3. Select a view from the menu or choose **Other...** for more views.

By default, the Debug perspective is opened with the following views:
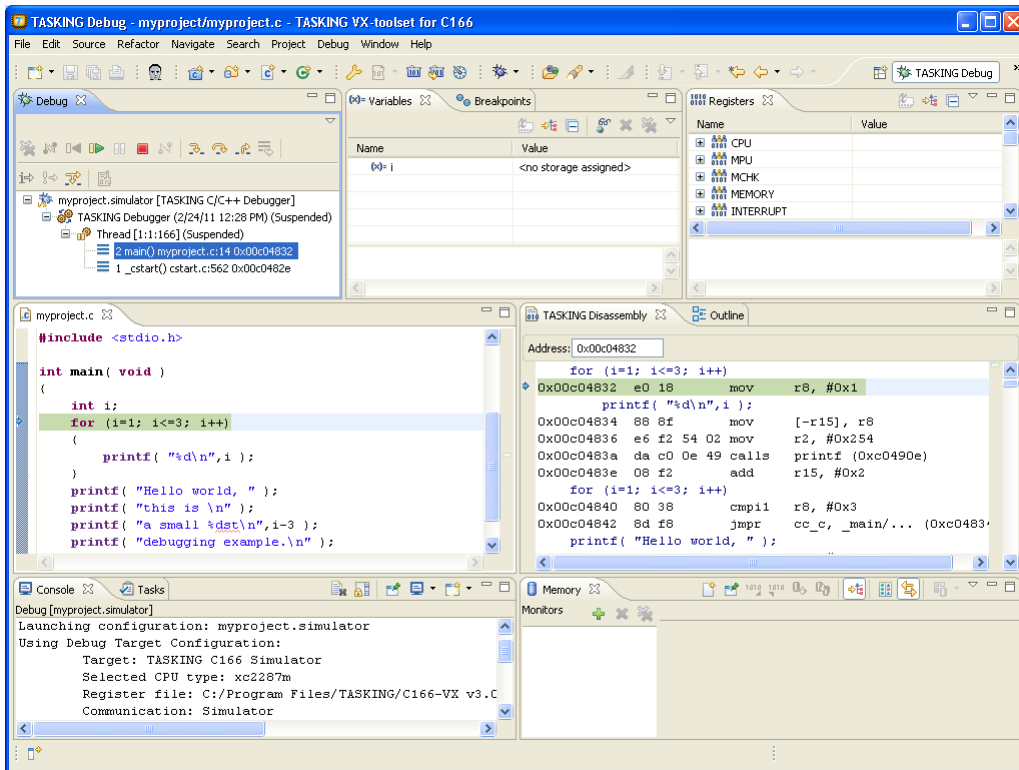
10.4.1. Debug View

The Debug view shows the target information in a tree hierarchy shown below with a sample of the possible icons:

Icon	Session item	Description
	Launch instance	Launch configuration name and launch type
	Debugger instance	Debugger name and state
	Thread instance	Thread number and state
	Stack frame instance	Stack frame number, function, file name, and file line number

The number beside the thread label is a reference counter, not a thread identification number (TID).

Stack display

During debugging (running) the actual stack is displayed as it increases or decreases during program execution. By default, all views present information that is related to the current stack item (variables, memory, source code etc.). To obtain the information from other stack items, click on the item you want.

The Debug view displays stack frames as child elements. It displays the reason for the suspension beside the thread, (such as end of stepping range, breakpoint hit, and signal received). When a program exits, the exit code is displayed.

The Debug view contains numerous functions for controlling the individual stepping of your programs and controlling the debug session. You can perform actions such as terminating the session and stopping the program. All functions are available from the right-click menu, though commonly used functions are also available from the toolbar in the Debug view.

Controlling debug sessions

Icon	Action	Description
	Remove all	Removes all terminated launches.
	Reset target system	Resets the target system and restarts the application.
	Resume	Resumes the application after it was suspended (manually, breakpoint, signal).
	Suspend	Suspends the application (pause). Use the Resume button to continue.
	Relaunch	Right-click menu. Restarts the selected debug session when it was terminated. If the debug session is still running, a new debug session is launched.
	Reload current application	Reloads the current application without restarting the debug session. The application does restart of course.
	Terminate	Ends the selected debug session and/or process. Use Relaunch to restart this debug session, or start another debug session.
	Terminate all	Right-click menu. As terminate. Ends *all* debug sessions.
	Terminate and remove	Right-click menu. Ends the debug session and removes it from the Debug view.
	Terminate and Relaunch	Right-click menu. Ends the debug session and relaunches it. This is the same as choosing Terminate end then Relaunch.
	Restart	Right-click menu. Restarts the application. The target system is *not* reset.
	Disconnect	Detaches the debugger from the selected process (useful for debugging attached processes)

Stepping through the application

Icon	Action	Description
	Step into	Steps to the next source line or instruction
	Step over	Steps over a called function. The function is executed and the application suspends at the next instruction after the call.
	Step return	Executes the current function. The application suspends at the next instruction after the return of the function.

Icon	Action	Description
i⇒	Instruction stepping	Toggle. If enabled, the stepping functions are performed on instruction level instead of on C source line level.
⇒	Interrupt aware stepping	Toggle. If enabled, the stepping functions do not step into an interrupt when it occurs.

Miscellaneous

Icon	Action	Description
📋	Copy Stack	Right-click menu. Copies the stack as text to the windows clipboard. You can paste the copied selection as text in, for example, a text editor.
🐞	Edit *project...*	Right-click menu. Opens the debug configuration dialog to let you edit the current debug configuration.
📝	Edit Source Lookup...	Right-click menu. Opens the Edit Source Lookup Path window to let you edit the search path for locating source files.

10.4.2. Breakpoints View

You can add, disable and remove breakpoints by clicking in the marker bar (left margin) of the Editor view. This is explained in the Getting Started manual.

Description

The Breakpoints view shows a list of breakpoints that are currently set. The button bar in the Breakpoints view gives access to several common functions. The right-most button ▽ opens the Breakpoints menu.

Types of breakpoints

To access the breakpoints dialog, add a breakpoint as follows:

1. Click the **Add TASKING Breakpoint** button (🔗).

   The Breakpoints dialog appears.

Each tab lets you set a breakpoint of a special type. You can set the following types of breakpoints:

- **File breakpoint**

  The target halts when it reaches the specified line of the specified source file. Note that it is possible that a source line corresponds to multiple addresses, for example when a header file has been included into two different source files or when inlining has occurred. If so, the breakpoint will be associated with all those addresses.

- **Function**

  The target halts when it reaches the first line of the specified function. If no source file has been specified and there are multiple functions with the given name, the target halts on all of those. Note that function breakpoints generally will not work on inlined instances of a function.

- **Address**

  The target halts when it reaches the specified instruction address.

- **Stack**

  The target halts when it reaches the specified stack level.

- **Data**

  The target halts when the given variable or memory location (specified in terms of an absolute address) is read or written to, as specified.

- **Instruction**

  The target halts when the given number of instructions has been executed.

- **Cycle**

  The target halts when the given number of clock cycles has elapsed.

- **Timer**

  The target halts when the given amount of time elapsed.

In addition to the type of the breakpoint, you can specify the condition that must be met to halt the program.

In the **Condition** field, type a condition. The condition is an expression which evaluates to 'true' (non-zero) or 'false' (zero). The program only halts on the breakpoint if the condition evaluates to 'true'.

In the **Ignore count** field, you can specify the number of times the breakpoint is ignored before the program halts. For example, if you want the program to halt only in the fifth iteration of a while-loop, type '4': the first four iterations are ignored.

10.4.3. File System Simulation (FSS) View

Description

The File System Simulation (FSS) view is automatically opened when the target requests FSS input or generates FSS output. The virtual terminal that the FSS view represents, follows the VT100 standard. If you right-click in the view area of the FSS view, a menu is presented which gives access to some self-explanatory functions.

VT100 characteristics

The queens example demonstrates some of the VT100 features. (You can find the queens example in the <C166 installation path>\examples directory from where you can import it into your workspace.) Per debugging session, you can have more than one FSS view, each of which is associated with a positive integer. By default, the view "FSS #1" is associated with the standard streams stdin, stdout, stderr and stdaux. Other views can be accessed by opening a file named "terminal window <number>", as shown in the example below.

```
FILE * f3 = fopen("terminal window 3", "rw");
fprintf(f3, "Hello, window 3.\n");
fclose(f3);
```

You can set the initial working directory of the target application in the Debug configuration dialog (see also Section 10.2, *Creating a Customized Debug Configuration*):

1. On the **Debugger** tab, select the **Miscellaneous** sub-tab.

2. In the **FSS root directory** field, specify the FSS root directory.

The FSS implementation is designed to work without user intervention. Nevertheless, there are some aspects that you need to be aware of.

First, the interaction between the C library code (in the files dbg*.c and dbg*.h; see Section 13.1.5, *dbg.h*) and the debugger takes place via a breakpoint, which incidentally is not shown in the Breakpoints view. Depending on the situation this may be a hardware breakpoint, which may be in short supply.

Secondly, proper operation requires certain code in the C library to have debug information. This debug information should normally be present but might get lost when this information is stripped later in the development process.

When you use MIL linking/splitting the C library is translated along with your application. Therefore you need to build your application with debug information generation enabled when FSS support is needed.

10.4.4. Disassembly View

The Disassembly view shows target memory disassembled into instructions and / or data. If possible, the associated C / C++ source code is shown as well. The **Address** field shows the address of the current selected line of code.

To view the contents of a specific memory location, type the address in the **Address** field. If the address is invalid, the field turns red.

10.4.5. Expressions View

The Expressions view allows you to evaluate and watch regular C expressions.

To add an expression:

Click **OK** to add the expression.

1. Right-click in the Expressions View and select **Add Watch Expression**.

   The Add Watch Expression dialog appears.

2. Enter an expression you want to watch during debugging, for example, the variable name "i"

If you have added one or more expressions to watch, the right-click menu provides options to **Remove** and **Edit** or **Enable** and **Disable** added expressions.

- You can access target registers directly using #NAME. For example "arr[#R0 << 3]" or "#TIMER3 = m++". If a register is memory-mapped, you can also take its address, for example, "&#ADCIN".

- Expressions may contain target function calls like for example "`g1 + invert(&g2)`". Be aware that this will not work if the compiler has optimized the code in such a way that the original function code does not actually exist anymore. This may be the case, for example, as a result of inlining. Also, be aware that the function and its callees use the same stack(s) as your application, which may cause problems if there is too little stack space. Finally, any breakpoints present affect the invoked code in the normal way.

10.4.6. Memory View

Use the Memory view to inspect and change process memory. The Memory view supports the same addressing as the C and C++ languages. You can address memory using expressions such as:

- `0x0847d3c`

- `(&y)+1024`

- `*ptr`

Monitors

To monitor process memory, you need to add a *monitor*:

1. In the Debug view, select a debug session. Selecting a thread or stack frame automatically selects the associated session.

2. Click the **Add Memory Monitor** button in the Memory Monitors pane.

   The Monitor Memory dialog appears.

3. Type the address or expression that specifies the memory section you want to monitor and click **OK**.

   The monitor appears in the monitor list and the Memory Renderings pane displays the contents of memory locations beginning at the specified address.

To remove a monitor:

1. In the Monitors pane, right-click on a monitor.

2. From the popup menu, select **Remove Memory Monitor**.

Renderings

You can inspect the memory in so-called *renderings*. A rendering specifies how the output is displayed: hexadecimal, ASCII, signed integer or unsigned integer. You can add or remove renderings per monitor. Though you cannot change a rendering, you can add or remove them:

1. Click the **New Renderings...** tab in the Memory Renderings pane.

   The Add Memory Rendering dialog appears.

2. Select the rendering you want (**Traditional**, **Hex**, **ASCII**, **Signed Integer**, **Unsigned Integer** or **Hex Integer**) and click **Add Rendering(s)**.

To remove a rendering:

1. Right-click on a memory address in the rendering.

2. From the popup menu, select **Remove Rendering**.

Changing memory contents

In a rendering you can change the memory contents. Simply type a new value.

> **Warning**: Changing process memory can cause a program to crash.

The right-click popup menu gives some more options for changing the memory contents or to change the layout of the memory representation.

10.4.7. Compare Application View

You can use the Compare Application view to check if the downloaded application matches the application in memory. Differences may occur, for example, if you changed memory addresses in the Memory view.

*   To check for differences, click the **Compare** button.

10.4.8. Heap View

With the Heap view you can inspect the status of the heap memory. This can be illustrated with the following example:

```
string = (char *) malloc(100);
strcpy ( string, "abcdefgh" );
free (string);
```

If you step through these lines during debugging, the Heap view shows the situation after each line has been executed. Before any of these lines has been executed, there is no memory allocated and the Heap view is empty.

*   After the first line the Heap view shows that memory is occupied, the description tells where the block starts, how large it is (`100 MAUs`) and what its content is (`0x0, 0x0, ...`).

*   After the second line, `"abcdefgh"` has been copied to the allocated block of memory. The description field of the Heap view again shows the actual contents of the memory block (`0x61, 0x62,...`).

*   The third line frees the memory. The Heap view is empty again because after this line no memory is allocated anymore.

10.4.9. Logging View

Use the Logging view to control the generation of internal log files. This view is intended mainly for use by or at the request of Altium support personnel.

10.4.10. RTOS View

The debugger has special support for debugging real-time operating systems (RTOSs). This support is implemented in an RTOS-specific shared library called a *kernel support module* (KSM) or *RTOS-aware debugging module* (RADM). Specifically, the TASKING VX-toolset for C166 ships with a KSM supporting the OSEK standard. You have to create your own OSEK Run Time Interface (ORTI) and specify this file on the **Miscellaneous** sub tab while configuring a customized debug configuration (see also Section 10.2, *Creating a Customized Debug Configuration*):

1. From the **Debug** menu, select **Debug Configurations...**

    The Debug Configurations dialog appears.

2. In the left pane, select the configuration you want to change, for example, **TASKING C/C++ Debugger » myproject.simulator**.

    Or: click the **New launch configuration** button ( ) to add a new configuration.

3. On the **Debugger** tab, select the **Miscellaneous** tab

4. In the **ORTI file** field, specify the name of your own ORTI file.

The debugger supports ORTI specifications v2.0 and v2.1.

10.4.11. Registers View

When first opened, the Registers view shows a number of *register groups*, which together contain all known registers. You can expand each group to see which registers they contain and examine the register's values while stepping through your application. This view has a number of features:

*   While you step through the application, the registers involved in the step turn yellow.

*   You can change each register's value.

*   You can copy registers and/or groups to the windows clipboard: select the groups and/or individual registers, right-click on a register(group) and from the popup menu choose **Copy Registers**. You can paste the copied selection as text in, for example, a text editor.

*   You can change the way the register value is displayed: right-click on a register(group) and from the popup menu choose the desired display mode (Natural, Hexadecimal, Decimal, Binary, Octal)

*   For some registers the menu entry **Symbolic Representation** is available in their right-click popup menu. This opens a new view which shows the internal fields of the register. For example, the CPUCON1 register from the CPU group may be shown as follows:
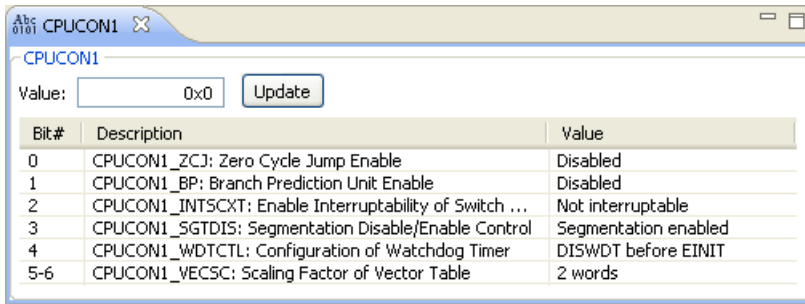
In this view you can set the individual values in the register, either by selecting a value from a drop-down box or by simply entering a value depending on the chosen field. To update the register with the new values, click the **Update** button.

- You can fully organize the register groups as you like: right-click on a register and from the popup menu use the menu items **Add Register Group...**, **Edit Register Group...** or **Remove Register Group**. This way you not only can choose which groups should be visible in the Register view, you can also create your own groups to which you add the registers of your interest.

  To restore the original groups: right-click on a register and from the popup menu choose **Restore Default Register Groups**. Be aware: groups you have created will be removed, groups you have edited are restored to their original and groups you have deleted are placed back!

Viewing a register group in a separate view

For a better overview, you can open a register group in a separate view. To do so, double-click on the register group name. A new Register view is opened, showing all registers from the group. You can consider this view as a sub view of the Register view with roughly the same features.

10.4.12. Trace View

If tracing is enabled, the Trace view shows the code that was most recently executed. For example, while you step through the application, the Trace view shows the executed code of each step. To enable tracing:

- From the **Debug** menu, select **Trace**.

  A check mark appears when tracing is enabled.

The view has three tabs, Source, Instruction and Raw, each of which represents the trace in a different way. However, not all target environments will support all three of these. The view is updated automatically each time the target halts.

10.5. Multi-Core Debug Support (MCDS)

The debugger contains several views offering access to Infineon Multi-Core Debug Support (MCDS) technology. You can use this only in conjunction with a hardware and software package sold by Hitex (for example the TantinoXC / XC2000 MCDS Extension). Consult Hitex' documentation for more information on installation and related aspects.

There are two debugger views that involve MCDS, each of which is discussed in the following sections.

10.5.1. Triggers - MCDS View

Triggers form one of the central concepts in MCDS. Much like a breakpoint, a trigger is a combination of a condition and an associated action.

To add a new trigger:

1. Right-click in the Triggers - MCDS view and from the context menu, select **New Trigger...**.
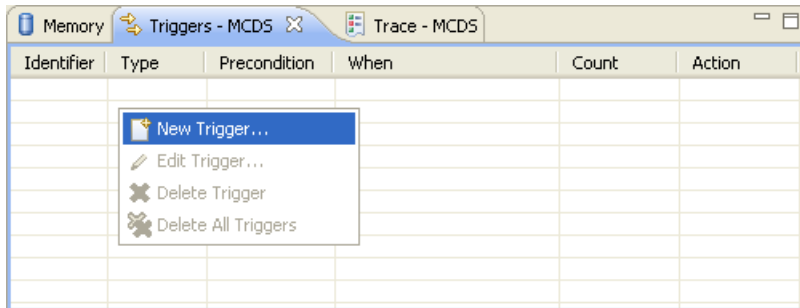
The New Trigger Definition dialog appears.
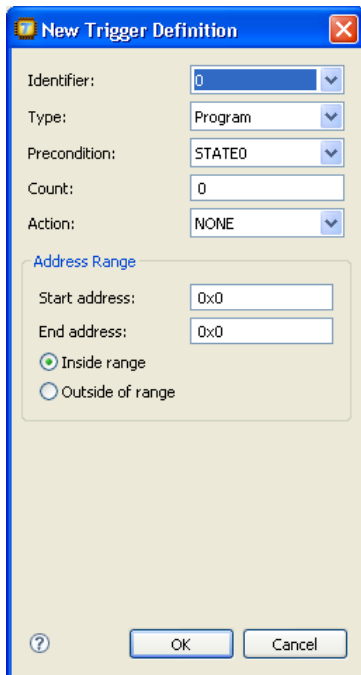
2. Specify a new trigger according to the fields described below and click **OK**.

- The **Identifier** field shows the trigger's identifier, which is a number between 0 and 15, reflecting the fact that the hardware supports a maximum of 16 triggers.

- In the **Type** box, you can select which processor activity should activate the trigger.

- Internally, the MCDS hardware contains a state flag, which can have the value 0 (default), 1 or 2. The **Precondition** box allows you to specify in which of these three states the trigger should be activated. (See **Action** below about how to switch states.)

  The value in the **Count** field defines how many times the trigger's activation should be skipped before the actual trigger is activated.

- The **Action** box specifies what action must be taken once the trigger's conditions have become true. You can choose between:

  - **NONE**. No action.

  - **Break**. Suspends execution (making the trigger resemble a regular breakpoint).

  - **TraceStop**. Stops tracing. See Section 10.5.2, *Trace - MCDS View*.

  - **TraceQualify**. A single frame will be recorded every time the trigger's conditions are met. See Section 10.5.2, *Trace - MCDS View*.

  - **STATE0,1,2**. Switches to a particular MCDS state (0, 1, or 2).

- The contents of the bottom half of the dialog depend on the selected type.

  For type **Program**, **ReadData** or **WriteData**, fill out the start address and end address (inclusive) of the address range and select whether the trigger should activate upon access inside or outside this range.

- For type **ReadData** or **WriteData**, you must also define the set of values that should activate the trigger. There are two options:

  - If the **Mask** option is not selected, the trigger will be activated if the value falls within the range spanned by the **Minimum value** and the **Maximum value**, or outside that range if **Inverted** is selected.

  - If the **Mask** option is selected, the logical AND of the value and the mask will be calculated. The trigger will be activated if this equals the **Minimum value** or does not equal this if **Inverted** is selected.

- For type **Combined**, select which triggers should be activated (or not activated) for the combined trigger itself to activate. Note that the triggers must be activated *simultaneously*.

For examples of several triggers, see the MCDS documentation of Hitex.

Keep in mind that in many cases the actual values read or written by the processor may differ from what is implied by the source code. For example, if you assign a value to a variable of type `unsigned long`, the compiler will typically split this up into two 16-bit writes, meaning that you cannot trigger on the 32-bit value.

10.5.2. Trace - MCDS View

MCDS makes it possible to record a trace of some or all of the following processor activities:

• the execution of instructions

• the reading of data from memory

• the writing of data to memory

To setup a trace:

1. Right-click in the Trace - MCDS view and from the context menu, select **Trace Setup...**.
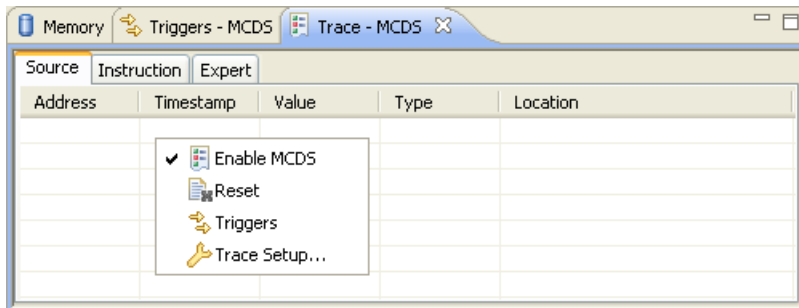
The MCDS Trace Setup dialog appears.
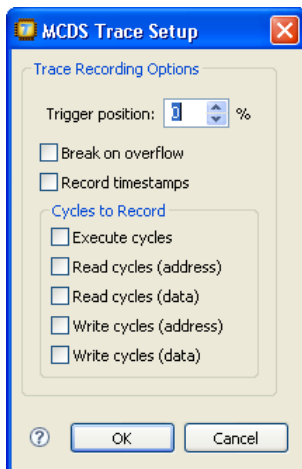
2. Specify the trace recording options according to the fields described below and click **OK**.

    • The **Trigger position** box controls functionality that is not yet available. You can ignore this field.

    • The **Break on overflow** option controls what happens when the trace buffer is full. If selected, MCDS will cause the target to halt. If not, the buffer will be overwritten cyclically, i.e. the oldest data will be discarded first.

    • With the **Record timestamps** option selected, the hardware will record a timestamp with frames, which will be shown in the view's **Time** column. Note however that for compression purposes, the hardware will only record a frame when the flow of execution changes. Consequently, some of the frames shown in the user interface may have been reconstructed by the software. No timestamp will be shown for these.

    • The options in the **Cycles to Record** box allow you to specify which processor activities to record.

Because of the finite size of the on-chip buffer in which trace information is stored, depending on the situation you may have to balance the length of the trace against the kind of information in it. For example, if you are investigating what location a suspect memory write is unintentionally affecting, you may want to select only the **Write cycles (address)** option, giving you a longer trace.

By default, the debugger records all processor activities as selected. However, with triggers you can narrow this down to, for example, a particular address range. By setting a particular trigger's **Action** to **TraceStop**, tracing stops at that point. Before doing this, deselect all five options in the **Cycles to Record** box. Despite these being deselected, execute, read and write cycles will all be logged until the trigger is activated. By setting it to **TraceQualify**, a single frame will be recorded every time the trigger's conditions are met. In this case, normal trace recording remains active as well. So, deselect the options in the **Cycles to Record** box if you only want to record the frames associated with the trigger.

10.6. Programming a Flash Device

With the TASKING debugger you can download an application file to flash memory. Before you download the file, you must specify the type of flash devices you use in your system and the address range(s) used by these devices.

To program a flash device the debugger needs to download a flash programming monitor to the target to execute the flash programming algorithm (target-target communication). This method uses temporary target memory to store the flash programming monitor and you have to specify a temporary data workspace for interaction between the debugger and the flash programming monitor.

Two types of flash devices can exist: on-chip flash devices and external flash devices.

Setup an on-chip flash device

When you specify a target configuration board using the Import Board Configuration wizard, as explained in the *Getting Started with the TASKING VX-toolset for C166* manual, any on-chip flash devices are setup automatically.

Setup an external flash device

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

    The Properties for project dialog appears.

2. In the left pane, expand **Run/Debug Settings** and select **Flash Programming**.

    The Flash Programming pane appears.
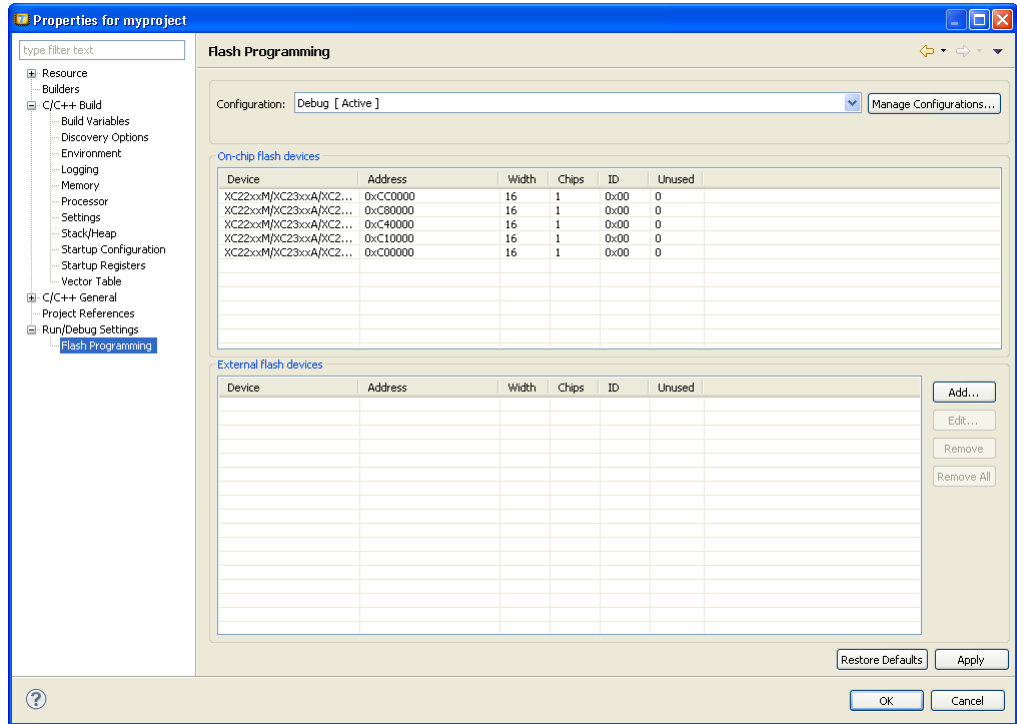
3. Click **Add...** to specify an external flash device.

    The Select a New Flash Device dialog appears.
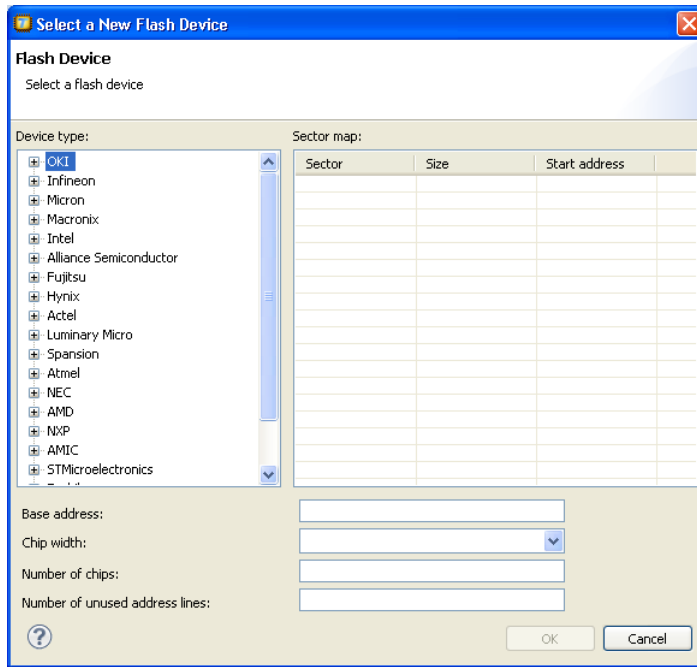
4.   In the **Device type** box, expand the name of the manufacturer of the device and select a device.

      The Sector map displays the memory layout of the flash device(s). Each sector has a size and

5.   In the **Base address** field enter the start address of the memory range that will be covered by the flash device. Any following addresses separated by commas are considered mirror addresses. This allows the flash device to be programmed through its mirror address before switching the flash to its base address.

6.   In the **Chip width** field select the width of the flash device.

7.   In the **Number of chips** field, enter the number of flash devices that are located in parallel. For example, if you have two 8-bit devices in parallel attached to a 16-bit data bus, enter 2.

8.   Fill in the **Number of unused address lines** field, if necessary.

The flash memory is added to the linker script file automatically with the tag "`flash=`*flash-id*".

To program a flash device

1.   From the **Debug** menu, select **Debug Configurations...**

      The Debug Configurations dialog appears.

2.   In the left pane, select the configuration you want to change, for example, **TASKING C/C++ Debugger » myproject.board**.

3. On the **Debugger** tab, select the **Initialization** tab

4. Enable the option **Program flash when downloading**.

   The Flash settings group box becomes active.

5. In the **Monitor file** field, specify the filename of the flash programming monitor, usually an Intel Hex or S-Record file.

6. In the **Sector buffer size** field, specify the buffer size for buffering a flash sector.

7. Specify the data **Workspace address** used by the flash programming monitor. This address may not conflict with the addresses of the flash devices.

8. Click **Debug** to program the flash device and start debugging.

Chapter 11. Target Board Support

This chapter contains an overview of the target boards (sorted by manufacturer) that are currently supported by the TASKING VX-toolset for C166. For each board is listed if you need to import special settings for this board into your project with the Import Board Configuration wizard. For example, a board can have extra memory or flash you want to use in your project.

11.1. Overview of Supported Boards

FS-FORTH Systeme

Name	Processor	Import board configuration
FS-FORTH EVA167 with C167CR Evaluation Board	C167CR	yes
FS-FORTH STart27x (272) Evaluation Board	ST10F272E	yes
FS-FORTH STart27x (276) Evaluation Board	ST10F276E	yes

Infineon

Name	Processor	Import board configuration
Infineon XC16x Easykit Board	XC16x	no
Infineon XC16x Starter Kit Board	XC16x	yes
Infineon XC164CM U CAN	XC164CM-8FF	no
Infineon XC2XXX/XE166 Easykit Board	XC2xxx/XE16x	no
Infineon XE164F U Connect	XE164F-96F	yes

I+ME ACTIA

Name	Processor	Import board configuration
I+ME C167C Evaluation Board	C167	yes

Phytec

Name	Processor	Import board configuration
Phytec kitCON-161V/K/O	C161O	yes
Phytec kitCON-161CS/CI/JC	C161CS/CI/JC	yes
Phytec KitCON-161PI/RI	C161PI/RI	yes
Phytec kitCON-163	C163	yes
Phytec kitCON-164	C164CI	yes
Phytec kitCON-165	C165	yes
Phytec kitCON-167	C167CR	yes

Name	Processor	Import board configuration
Phytec MiniModule-165	C165	yes
Phytec miniMODUL-167	C167CR/CS	yes
Phytec nanoMODUL-164	C164CI	yes
Phytec phyCORE-161CS/JC/JI	C161CS/JC/JI	yes
Phytec phyCORE-167CR	C167CR/CS	yes
Phytec phyCORE-ST10F276	ST10F276E	yes
Phytec phyCORE-XC161CJ	XC161CJ-16FF	yes
Phytec phyCORE-XC167CI	XC167CI	yes

STMicroelectronics

Name	Processor	Import board configuration
STMicroelectronics ST10F252 Evaluation Board	ST10F252E	yes

TKtronic

Name	Processor	Import board configuration
TKtronic TKeva164 Evaluation Board	C164CI	yes

TQ-Components

Name	Processor	Import board configuration
TQ-Components STK164C	C164CI	yes
TQ-Components STKx167U	XC167CI	yes
TQ-Components STKx161U	XC161CJ-16FF	yes
TQ-Components STK167C/LC/U/UE/UL	C167CR/CS	yes

Chapter 12. Tool Options

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

Tool options in Eclipse (Menu entry)

For each tool option that you can set from within Eclipse, a **Menu entry** description is available. In Eclipse you can customize the tools and tool options in the following dialog:

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

   The Properties dialog appears.

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

   In the right pane the Settings appear.

3. Open the **Tool Settings** tab.

   You can set all tool options here.

Unless stated otherwise, all **Menu entry** descriptions expect that you have this Tool Settings tab open.

12.1. Configuring the Command Line Environment

If you want to use the tools on the command line (either using a Windows command prompt or using Solaris), you can set *environment variables*.

You can set the following environment variables:

Environment variable	Description
AS166INC	With this variable you specify one or more additional directories in which the assembler looks for include files. See Section 7.3, *How the Assembler Searches Include Files*.
C166INC	With this variable you specify one or more additional directories in which the C compiler looks for include files. See Section 4.4, *How the Compiler Searches Include Files*.
CP166INC	With this variable you specify one or more additional directories in which the C++ compiler looks for include files. See Section 5.2, *How the C++ Compiler Searches Include Files*.
CC166BIN	When this variable is set, the control program prepends the directory specified by this variable to the names of the tools invoked.

Environment variable	Description
LIBC166	With this variable you specify one or more additional directories in which the linker looks for libraries. See Section 8.3.1, *How the Linker Searches Libraries*.
LM_LICENSE_FILE	With this variable you specify the location of the license data file. You only need to specify this variable if the license file is not on its default location (`c:\flexlm\license.dat` for Windows, `/usr/local/flexlm/licenses/license.dat` for Solaris).
PATH	With this variable you specify the directory in which the executables reside. This allows you to call the executables when you are not in the `bin` directory. Usually your system already uses the PATH variable for other purposes. To keep these settings, you need to add (rather than replace) the path. Use a semicolon (;) to separate path names.
TASKING_LIC_WAIT	If you set this variable, the tool will wait for a license to become available, if all licenses are taken. If you have not set this variable, the tool aborts with an error message. (Only useful with floating licenses)
TMPDIR	With this variable you specify the location where programs can create temporary files. Usually your system already uses this variable. In this case you do not need to change it.

See the documentation of your operating system on how to set environment variables.

12.2. C Compiler Options

This section lists all C compiler options.

Options in Eclipse versus options on the command line

Most command line options have an equivalent option in Eclipse but some options are only available on the command line. Eclipse invokes the compiler via the control program. Therefore, it uses the syntax of the control program to pass options and files to the C compiler. If there is no equivalent option in Eclipse, you can specify a command line option in Eclipse as follows:

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

    The Properties dialog appears.

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

    In the right pane the Settings appear.

3. On the Tool Settings tab, select **C/C++ Compiler » Miscellaneous**.

4. In the **Additional options** field, enter one or more command line options.

    *Because Eclipse uses the control program, you have to precede the option with **-Wc** to pass the option via the control program directly to the C compiler.*

Be aware that some command line options are not useful in Eclipse or just do not have any effect. For example, the option **-n** sends output to stdout instead of a file and has no effect in Eclipse.

Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (**-**) character, long option names always begin with two minus (**--**) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

Options can have flags or suboptions. To switch a flag 'on', use a lowercase letter or a **+***longflag*. To switch a flag off, use an uppercase letter or a **-***longflag*. Separate *longflags* with commas. The following two invocations are equivalent:

```
c166 -Oac test.c
c166 --optimize=+coalesce,+cse test.c
```

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

C compiler option: --alternative-sfr-file

Menu entry

1.   Select **C/C++ Compiler » Preprocessing**.

2.   Enable the option **Use alternative SFR file format**.

Command line syntax

```
--alternative-sfr-file
```

Description

With this option the C compiler includes the alternative SFR file without alias definitions for SFRs and SFR bit-fields. This alternative SFR file is named reg*cpu*.asfr and is located in the product's include/sfr directory.

Use this option to speed up your compilation (smaller SFR file) and have less namespace pollution.

Related information

Section 1.3.5, *Accessing Hardware from C*

C compiler option: --automatic-near

Menu entry

1. Select **C/C++ Compiler » Allocation**.

2. Enable the option **Application wide automatic near data allocation**.

Command line syntax

```
--automatic-near
```

Description

With this option you enable automatic near data allocation, where default pointer qualifiers may be replaced by more optimal qualifiers.

This optimization can only be used together with the MIL linking or MIL splitting build process, because it needs application scope.

Related information

Section 4.6.2, *Core Specific Optimizations (backend)*

C compiler option **--mil** / **--mil-split**

C compiler option: --bita-struct-threshold

Menu entry

1. Select **C/C++ Compiler » Allocation**

2. In the **Threshold for putting structs in __bita** field, enter a value in bytes.

Command line syntax

`--bita-struct-threshold=`*threshold*

Default: `--bita-struct-threshold=0`

Description

With this option the compiler allocates unqualified structures that are smaller than or equal to the specified *threshold* and have a bit-field of length one in the bit-addressable (`__bita`) memory space automatically. The *threshold* must be specified in bytes. Objects that are qualified `const` or `volatile` or objects that are absolute are not moved.

By default the *threshold* is 0 (off), which means that no objects are moved.

Note that this option has no effect when you use MIL linking/splitting (**--mil/--mil-split**).

Example

To put all unqualified structures with a size of 4 bytes or smaller and a bit-field of length one into the `__bita` section:

```
c166 --bita-struct-threshold=4 test.c
```

Related information

C compiler option **--mil** / **--mil-split**

C compiler option: --cert

Menu entry

1. Select **C/C++ Compiler » CERT C Secure Coding**.

2. Make a selection from the **CERT C secure code checking** list.

3. If you selected **Custom**, expand the **Custom CERT C** entry and enable one or more individual recommendations/rules.

Command line syntax

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

Default format: all

Description

With this option you can enable one or more checks for CERT C Secure Coding Standard recommendations/rules. When you omit the argument, all checks are enabled. *name* is the name of a CERT recommendation/rule, consisting of three letters and two digits. Specify only the three-letter mnemonic to select a whole category. For the list of names you can use, see Chapter 19, *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:

```
c166 --cert=str30 test.c
```

Related information

Chapter 19, *CERT C Secure Coding Standard*

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

C compiler option: --check

Menu entry

-

Command line syntax

`--check`

Description

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

The compiler reports any warnings and/or errors.

This option is available on the command line only.

Related information

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

C compiler option: --compact-max-size

Menu entry

1. Select **C/C++ Compiler » Optimization**.

2. In the **Maximum size for code compaction** field, enter the maximum size of a match.

Command line syntax

`--compact-max-size=`*value*

Default: 200

Description

This option is related to the compiler optimization **--optimize=+compact** (Code compaction or reverse inlining). Code compaction is the opposite of inlining functions: large sequences of code that occur more than once, are transformed into a function. This reduces code size (possibly at the cost of execution speed).

However, in the process of finding sequences of matching instructions, compile time and compiler memory usage increase quadratically with the number of instructions considered for code compaction. With this option you tell the compiler to limit the number of matching instructions it considers for code compaction.

Example

To limit the maximum number of instructions in functions that the compiler generates during code compaction:

```
c166 --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-memory

Menu entry

1.   Select **C/C++ Compiler » Allocation**.

2.   Select a **Memory space for constant values**.

Command line syntax

`--constant-memory=`*space*

Default: **--constant-memory=model**

Description

With this option you can control the allocation of constant values, automatic initializers and switch tables. The *space* must be one of `__near`, `__far`, `__shuge`, `__huge` or `model`.

The switch tables are never allocated in the spaces `__far` and `__huge`; `__shuge` will be used instead.

By default the compiler allocates constant values, automatic initializers and switch tables based on the selected memory model (C compiler option **--model**).

Variables that are qualified `const` are not affected by this option.

> When you use MIL splitting (option **--mil-split**) and automatic near data allocation (option **--automatic-near**), you cannot set the *space* to `__near`.

Example

```
int foo( void )
{
 int arr[] =  { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
 return arr[4];
}
```

The option **--constant-memory** controls the space of the section which contains the initializer `{ 1, 2, 3, 4, 5, 6, 7, 8, 9 }`.

```
long long bar( void )
{
 return( 0x1234567812345678 );
}
```

The option **--constant-memory** controls the space of the section which contains the `long long` constant `0x1234567812345678`.

To compile the file `test.c` for the far memory model, but allocate constant values in `__near`:

```
c166 --model=far --constant-memory=__near test.c
```

Related information

Pragma `constant_memory`

C compiler option **--model** (Select memory model)

C compiler option: --core

Menu entry

1. Expand **C/C++ Build** and select **Processor**.

2. From the **Processor selection** list, expand **Custom** and select a core.

Command line syntax

`--core=`*core*

You can specify the following *core* arguments:

c16x	C16x instruction set
st10	ST10 instruction set
st10mac	ST10 with MAC co-processor support
c166sv1	C166S v1.0 support
xc16x	XC16x/XE16xx/XC2xxx instruction set
super10	Super10 instruction set
super10m345	Enhanced Super10M345 instruction set

Default: derived from `--cpu` if specified and known or else `--core=xc16x`

Description

With this option you specify the core architecture for a target processor for which you create your application. If **--cpu** is specified and the supplied CPU is known by the C compiler, the C compiler selects the correct core automatically.

For more information see C compiler option **--cpu**.

Example

Specify a custom core:

```
c166 --core=st10 test.c
```

Related information

C compiler option **--cpu** (Select processor)

C compiler option: --cpu (-C)

Menu entry

1.   Expand **C/C++ Build** and select **Processor**.

2.   From the **Processor selection** list, select a processor or expand **Custom** and select a core.

Command line syntax

--cpu=`cpu`

-C`cpu`

Description

With this option you define the target processor for which you create your application.

Based on this option the compiler always includes the special function register file reg`cpu`.sfr or reg`cpu`.asfr, unless you specify option **--no-tasking-sfr**.

If you select a target from the list, the core is known. If you specify a Custom processor, you need to select the core that matches the core of your custom processor (option **--core**). The C compiler knows all CPUs with core c16x, st10, st10mac and super10. If you specify a CPU that is not in that list, it is assumed to be an xc16x.

The following table show the relation between the two options:

--cpu=*cpu*	--core=*core*	*core*	Register file
no	no	c16x	c166, cp166: none as166: regc16x.sfr
no	yes	*core*	c166, cp166: none as166: reg*core*.sfr
yes	no	derived from *cpu* for core c16x, st10, st10mac and super10 If the *cpu* is unknown core xc16x is assumed	reg*cpu*.sfr
yes	yes	*core*	reg*cpu*.sfr

The standard list of supported processors is defined in the file processors.xml. This file defines for each processor its full name (for example, XC2287-72F), the base CPU name (for example, xc2287), the core settings (for example, xc16x), the on-chip flash settings, the list of silicon bugs for that processor. Each processor also defines an option to supply to the linker for preprocessing the LSL file for the applicable on-chip memory definitions. The option is for example **-DXC2287_72M**.

Example

Specify an existing processor:

```
c166 --cpu=c167cs40 test.c
```

Specify a custom processor:

```
c166 --cpu=custom --core=st10 test.c
```

Related information

C compiler option **--core** (Select the core)

C compiler option **--no-tasking-sfr** (Do not include SFR file)

C compiler option **--alternative-sfr-file** (Use alternative SFR file format)

Section 1.3.5, *Accessing Hardware from C*

C compiler option: --debug-info (-g)

Menu entry

1. Select **C/C++ Compiler » Debugging**.

2. To **generate symbolic debug information**, select **Default**, **Small set** or **Full**.
    To disable the generation of debug information, select **None**.

Command line syntax

`--debug-info`[**`=`**_suboption_]

`-g`[_suboption_]

You can set the following suboptions:

small	**1** / **c**	Emit small set of debug information.
default	**2** / **d**	Emit default symbolic debug information.
all	**3** / **a**	Emit full symbolic debug information.

Default: **`--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.

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.

Related information

-

C compiler option: --define (-D)

Menu entry

1.   Select **C/C++ Compiler » Preprocessing**.

     The Defined symbols box shows the symbols that are currently defined.

2.   To define a new symbol, click on the **Add** button in the **Defined symbols** box.

3.   Type the symbol definition (for example, demo=1)

Use the **Edit** and **Delete** button to change a macro definition or to remove a macro from the list.

Command line syntax

--define=_macro_name_[**=**_macro_definition_]

-D_macro_name_[**=**_macro_definition_]

Description

With this option you can define a macro and specify it to the preprocessor. If you only specify a macro name (no macro definition), the macro expands as '1'.

You can specify as many macros as you like. Simply use the **Add** button to add new macro definitions.

On the command line, you can use the option **--define** (**-D**) multiple times. If the command line exceeds the limit of the operating system, you can define the macros in an option file which you then must specify to the compiler with the option **--option-file** (**-f**) *file*.

Defining macros with this option (instead of in the C source) is, for example, useful to compile conditional C source as shown in the example below.

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:

```
c166 --define=DEMO test.c
c166 --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.

```
c166 --define="MAX(A,B)=((A) > (B) ? (A) : (B))" test.c
```

Related information

C compiler option **--undefine** (Remove preprocessor macro)

C compiler option **--option-file** (Specify an option file)

C compiler option: --dep-file

Menu entry

Eclipse uses this option in the background to create a file with extension `.d` (one for every input file).

Command line syntax

--dep-file[**=***file*]

Description

With this option you tell the compiler to generate dependency lines that can be used in a Makefile. In contrast to the option **--preprocess=+make**, the dependency information will be generated in addition to the normal output file.

By default, the information is written to a file with extension `.d` (one for every input file). When you specify a filename, all dependencies will be combined in the specified file.

Example

```
c166 --dep-file=test.dep test.c
```

The compiler compiles the file `test.c`, which results in the output file `test.src`, and generates dependency lines in the file `test.dep`.

Related information

C compiler option **--preprocess=+make** (Generate dependencies for make)

C compiler option: --diag

Menu entry

1. From the **Window** menu, select **Show View » Other » Basic » Problems**.

    The Problems view is added to the current perspective.

2. In the Problems view right-click on a message.

    A popup menu appears.

3. Select **Detailed Diagnostics Info**.

    A dialog box appears with additional information.

Command line syntax

--diag=[*format*:]{**all** | *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:

```
c166 --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:

```
c166 --diag=html:all > cerrors.html
```

Related information

Section 4.9, *C Compiler Error Messages*

C compiler option **--cert** (Enable individual CERT checks)

C compiler option: --dwarf-encoding

Menu entry

-

Command line syntax

--dwarf-encoding=_version_

Default: **--dwarf-encoding=2**

Description

The TASKING C166 ELF/DWARF ABI document describes the implementation of the ELF object format and the DWARF 3 debug information for the TASKING VX-toolset for C166. Within the DWARF 3 debug information two encoding versions are available to specify which TASKING type qualifier extensions are available (see section 2.3 in the EDABI). With this option you can set the version of the DWARF 3 encoding the C compiler should use. The TASKING debugger uses encoding version 2, which means that also the qualifiers __bita and __iram are supported. If you use a (third party) debugger that uses the older DWARF encoding, you can set the encoding version to 1.

Related information

Section 15.1, *ELF/DWARF Object Format*

Section 2.3 in the latest TASKING C166 ELF/DWARF ABI document at
http://www.altium.com/TASKING/support/c166/

C compiler option: --error-file

Menu entry

-

Command line syntax

--error-file[**=***file*]

Description

With this option the compiler redirects error messages to a file. If you do not specify a filename, the error file will be named after the input file with extension `.err`.

Example

To write errors to `errors.err` instead of `stderr`, enter:

```
c166 --error-file=errors.err test.c
```

Related information

-

C compiler option: --global-type-checking

Menu entry

1.   Select **C/C++ Compiler » Diagnostics**.

2.   Enable the option **Perform global type checking on C code**.

Command line syntax

```
--global-type-checking
```

Description

The C compiler already performs type checking within each module. Use this option when you want the linker to perform type checking between modules.

Related information

-

C compiler option: --help (-?)

Menu entry

-

Command line syntax

--help[**=***item*]

-?

You can specify the following arguments:

intrinsics	**i**	Show the list of intrinsic functions
options	**o**	Show extended option descriptions
pragmas	**p**	Show the list of supported pragmas
typedefs	**t**	Show the list of predefined typedefs

Description

Displays an overview of all command line options. With an argument you can specify which extended information is shown.

Example

The following invocations all display a list of the available command line options:

```
c166 -?
c166 --help
c166
```

The following invocation displays a list of the available pragmas:

```
c166 --help=pragmas
```

Related information

-

C compiler option: --ignore-memory-type-for-automatics

Menu entry

-

Command line syntax

`--ignore-memory-type-for-automatics`

Description

Normally the C compiler does not accept a memory type other than `__near` for automatic/parameter variables. With this option any memory type is accepted, but the C compiler ignores it and uses `__near` instead.

Related information

-

C compiler option: --include-directory (-I)

Menu entry

1.   Select **C/C++ Compiler » Include Paths**.

     The Include paths box shows the directories that are added to the search path for include files.

2.   To define a new directory for the search path, click on the **Add** button in the **Include paths** box.

3.   Type or select a path.

Use the **Edit** and **Delete** button to change a path or to remove a path from the list.

Command line syntax

--include-directory=*path*,...

-I*path*,...

Description

With this option you can specify the path where your include files are located. A relative path will be relative to the current directory,

The order in which the compiler searches for include files is:

1. The pathname in the C source file and the directory of the C source (only for #include files that are enclosed in "")

2. The path that is specified with this option.

3. The path that is specified in the environment variable `C166INC` 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:

```
c166 --include-directory=myinclude test.c
```

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

The compiler now looks for the file `myinc.h` in the directory where `test.c` is located. If the file is not there the compiler searches in the directory `myinclude`. If it was still not found, the compiler searches in the environment variable and then in the default include directory.

Related information

C compiler option **--include-file** (Include file at the start of a compilation)

C compiler option **--no-stdinc** (Skip standard include files directory)

C compiler option: --include-file (-H)

Menu entry

1. Select **C/C++ Compiler » Preprocessing**.

   The Pre-include files box shows the files that are currently included before the compilation starts.

2. To define a new file, click on the **Add** button in the **Pre-include files** box.

3. Type the full path and file name or select a file.

   Use the **Edit** and **Delete** button to change a file name or to remove a file from the list.

Command line syntax

`--include-file=`*file*`,...`

`-H`*file*`,...`

Description

With this option you include one or more extra files at the beginning of each C source file, before other includes. This is the same as specifying `#include "file"` at the beginning of *each* of your C sources.

Example

`c166 --include-file=stdio.h test1.c test2.c`

The file `stdio.h` is included at the beginning of both `test1.c` and `test2.c`.

Related information

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

C compiler option: --inline

Menu entry

1.   Select **C/C++ Compiler » Optimization**.

2.   Enable the option **Always inline function calls**.

Command line syntax

```
--inline
```

Description

With this option you instruct the compiler to inline calls to functions without the `__noinline` function qualifier whenever possible. This option has the same effect as a `#pragma inline` at the start of the source file.

This option can be useful to increase the possibilities for code compaction (C compiler option **--optimize=+compact**).

Example

To always inline function calls:

```
c166 --optimize=+compact --inline test.c
```

Related information

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

Section 1.12.3, *Inlining Functions: inline*

C compiler option: --inline-max-incr / --inline-max-size

Menu entry

1. Select **C/C++ Compiler » Optimization**.

2. In the **Maximum size increment when inlining** field, enter a value (default -1).

3. In the **Maximum size for functions to always inline** field, enter a value (default -1).

Command line syntax

```
--inline-max-incr=percentage  (default: -1)

--inline-max-size=threshold   (default: -1)
```

Description

With these options you can control the automatic function inlining optimization process of the compiler. These options 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

```
c166 --inline-max-incr=40 --inline-max-size=15 test.c
```

The compiler first inlines all functions with the function qualifier `inline` and all functions that are smaller than the specified threshold of 15. If the code size has still not increased with 40%, the compiler decides which other functions it can inline.

Related information

C compiler option **--optimize=+inline** (Optimization: automatic function inlining)

Section 1.12.3, *Inlining Functions: inline*

Section 4.6.3, *Optimize for Size or Speed*

C compiler option: --integer-enumeration

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Treat enumerated types always as integer**.

Command line syntax

`--integer-enumeration`

Description

Normally the compiler treats enumerated types as the smallest data type possible (`char` instead of `int`). This reduces code size. With this option the compiler always treats enum types as `int` as defined in the ISO C99 standard.

Related information

Section 1.1, *Data Types*

C compiler option: --iso (-c)

Menu entry

1. Select **C/C++ Compiler » Language**.

2. From the **Comply to C standard** list, select **ISO C99** or **ISO C90**.

Command line syntax

--iso={90|99}

-c{90|99}

Default: **--iso=99**

Description

With this option you select the ISO C standard. C90 is also referred to as the "ANSI C standard". C99 refers to the newer ISO/IEC 9899:1999 (E) standard. C99 is the default.

Example

To select the ISO C90 standard on the command line:

```
c166 --iso=90 test.c
```

Related information

C compiler option **--language** (Language extensions)

C compiler option: --keep-output-files (-k)

Menu entry

Eclipse *always* removes the `.src` file when errors occur during compilation.

Command line syntax

`--keep-output-files`

`-k`

Description

If an error occurs during compilation, the resulting `.src` file may be incomplete or incorrect. With this option you keep the generated output file (`.src`) when an error occurs.

By default the compiler removes the generated output file (`.src`) when an error occurs. This is useful when you use the make utility. If the erroneous files are not removed, the make utility may process corrupt files on a subsequent invocation.

Use this option when you still want to inspect the generated assembly source. Even if it is incomplete or incorrect.

Example

`c166 --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)

383

C compiler option: --language (-A)

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable or disable one or more of the following options:

   - Allow 32/16 -> 16 bit division and modulo

   - Use 14 bits arithmetic for far pointer comparison

   - Allow GNU C extensions

   - Allow // comments in ISO C90 mode

   - Check assignment of string literal to non-const string pointer

   - Allow optimization across volatile access

Command line syntax

--language=[*flags*]

-A[*flags*]

You can set the following flags:

+/-div32	**d/D**	32/16 -> 16-bit division and modulo instructions
+/-cmp14	**f/F**	14-bit arithmetic for far pointer comparison
+/-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: **-AdfGpVx**

Default (without flags): **-ADFGPVX**

Description

With this option you control the language extensions the compiler can accept. By default the C166 compiler allows all language extensions, except for **gcc** extensions.

The option **--language** (**-A**) without flags disables all language extensions.

32/16 -> 16 bit division and modulo instructions

When a 32 bit value is divided by a 16 bits divisor and only 16 bits of the result are being used, then the operation can be done by a DIVL or DIVLU instruction, depending on the signed/unsigned setting of the

operands. The same applies for the modulo operator. When there are chances for overflow and the (truncated) result must still be conform ISO C99, then it is better to switch this option off.

The next example generates a DIVL instruction when compiled with **--language=+div32**:

```
long m32;
short m16,divisor;

short div3216(long lm32,short ldiv)
{
    return (short)(lm32/ldiv);
}

int main (void)
{
    m32=1000;
    divisor=250;
    m16=div3216(m32,divisor);
    return m16;
}
```

See also the intrinsic functions __div32, __divu32, __mod32 and __modu32.

14-bit arithmetic for far pointer comparison

With **--language=+cmp14** (**-Af**) you tell the compiler to allow 14-bit arithmetic for far pointer comparison. 14-bit arithmetic is used for far pointer comparison instead of long 32-bit arithmetic. Only the page offset is compared. Far pointers do not cross page boundaries and if the objects pointing to are not members of the same aggregate or (union) object, the result is undefined.

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"; }
```

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   .proc   far
    movw    r11,_variable  ; <== Moved across volatile access
    movw    _access,ZEROS  ; <== Volatile access
    orw     r11,r2
    cmpw    r11,#0x3
    jmp     cc_ne,_2
    movw    r11,#0x0
_2:
    bset    r11.15
    movw    r12,#0x1
    movw    _access,r12    ; <== Volatile access
    movw    _variable,r11  ; <== Moved across volatile access
    ret
```

Result with **--language=+volatile**:

```
_TestFunc   .proc   far
    movw    _access,ZEROS  ; <== Volatile access
    orw     _variable,r2
    movw    r11,#0x3
    cmpw    r11,_variable
    jmp     cc_ne,_2
    movw    _variable,ZEROS
_2:
    movw    r11,#0x8000
    orw     _variable,r11
    movw    r11,#0x1
    movw    _access,r11    ; <== Volatile access
    ret
```

Note that the volatile behavior of the compiler with option **--language=-volatile** or **--language=+volatile** is ISO C compliant in both cases.

Example

```
c166 --language=-comments,+strings --iso=90 test.c
c166 -APx -c90 test.c
```

The compiler compiles in ISO C90 mode, accepts assignments of a constant string to a non-constant string pointer and does not allow C++ style comments.

Related information

C compiler option **--iso** (ISO C standard)

Section 1.16, *TASKING Volatile Implementation*

C compiler option: --lsl-define

Menu entry

-

Command line syntax

`--lsl-define=`*macro_name*[`=`*macro_definition*]`,...`

Description

With this option you can define a macro and specify it to the LSL file preprocessor. If you only specify a macro name (no macro definition), the macro expands as '1'.

Related information

Linker option **--define** (Define linker script file macro)

Section 8.7, *Controlling the Linker with a Script*

C compiler option: --lsl-file

Menu entry

1. Select **C/C++ Compiler » Optimization**.

2. Enable the option **Build for application wide optimizations (MIL linking)**.

3. Select **Optimize less/Build faster** or **Optimize more/Build slower**.

4. Select **C/C++ Compiler » Allocation**.

5. Enable the option **Automatic near data allocation**.

Command line syntax

```
--lsl-file=file,...
```

Description

With this option you specify one or more linker script files to the C compiler. The linker script file is used during the MIL linking phase of the build process, in the automatic near data allocation stage.

Related information

C compiler option **--mil** / **--mil-split**

C compiler option **--automatic-near** (Automatic near data allocation)

Linker option **--lsl-file** (Linker script file)

Section 4.1, *Compilation Process*

Section 8.7, *Controlling the Linker with a Script*

C compiler option: --lsl-include

Menu entry

-

Command line syntax

--lsl-include=`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 C compiler 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.`

Related information

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

C compiler option **--lsl-file** (Linker script file)

Section 8.7, *Controlling the Linker with a Script*

C compiler option: --lsl-strategy

Menu entry

-

Command line syntax

`--lsl-strategy=``strategy`

You can select the following strategy:

best-fit	**b**	Select a memory block where an object fits best. When blocks are equally suitable, select according to the locate direction in the LSL file.
direction	**d**	Select blocks according to the locate direction in the LSL file. This is the default.

Default: `--lsl-strategy=d`

Description

With this option you specify the locate strategy. During automatic near data allocation, objects need to be placed at absolute addresses. When you select the 'best-fit' strategy, this can lead to more optimal use of memory.

Related information

C compiler option **--automatic-near** (Automatic near data allocation)

Linker option **--lsl-file** (Linker script file)

Section 4.1, *Compilation Process*

Section 8.7, *Controlling the Linker with a Script*

C compiler option: --mac

Menu entry

1. Select **C/C++ Compiler » Code Generation**.

2. Enable the option **Automatic MAC code generation**.

Command line syntax

`--mac`

Description

With this option the compiler will try to use the MAC (multiply-accumulate) coprocessor automatically. This option does not affect objects that are qualified with the __mac keyword.

Related information

Section 1.14.2, *Manual MAC Qualification: __mac*

C compiler option **--no-savemac** (Do not save MAC registers)

C compiler option: --make-target

Menu entry

-

Command line syntax

`--make-target=``name`

Description

With this option you can overrule the default target name in the make dependencies generated by the options **--preprocess=+make** (**-Em**) and **--dep-file**. The default target name is the basename of the input file, with extension `.obj`.

Example

```
c166 --preprocess=+make --make-target=mytarget.obj test.c
```

The compiler generates dependency lines with the default target name `mytarget.obj` instead of `test.obj`.

Related information

C compiler option **--preprocess=+make** (Generate dependencies for make)

C compiler option **--dep-file** (Generate dependencies in a file)

C compiler option: --max-call-depth

Menu entry

1. Select **C/C++ Compiler » Optimization**.

2. In the **Maximum call depth for code compaction** field, enter a value.

Command line syntax

```
--max-call-depth=value
```

Default: -1

Description

This option is related to the compiler optimization **--optimize=+compact** (Code compaction or reverse inlining). Code compaction is the opposite of inlining functions: large sequences of code that occur more than once, are transformed into a function. This reduces code size (possibly at the cost of execution speed).

During code compaction it is possible that the compiler generates nested calls. This may cause the program to run out of its stack. To prevent stack overflow caused by too deeply nested function calls, you can use this option to limit the call depth. This option can have the following values:

-1	Poses no limit to the call depth (default)
0	The compiler will not generate any function calls. (Effectively the same as if you turned 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

Menu entry

1. Select **C/C++ Compiler » Optimization**.

2. Enable the option **Build for application wide optimizations (MIL linking)**.

3. Select **Optimize less/Build faster** or **Optimize more/Build slower**.

Command line syntax

```
--mil
--mil-split[=file,...]
```

Description

With option **--mil** the C compiler skips the code generator phase and writes the optimized intermediate representation (MIL) to a file with the suffix .mil. The C compiler accepts .mil files as input files on the command line.

Option **--mil-split** does the same as option **--mil**, but in addition, the C compiler splits the MIL representation and writes it to separate files with suffix .ms. One file is written for each input file or MIL library specified on the command line. The .ms files are only updated on a change. The C compiler accepts .ms files as input files on the command line.

With option **--mil-split** you can perform application-wide optimizations during the frontend phase by specifying all modules at once, and still invoke the backend phase one module at a time to reduce the total compilation time. Application wide code compaction is not possible in this case.

Optionally, you can specify another filename for the .ms file the C compiler generates. Without an argument, the basename of the C source file is used to create the .ms filename. Note that if you specify a filename, you have to specify one filename for every input file.

Build for application wide optimizations (MIL linking) and Optimize less/Build faster

This option is standard MIL linking and splitting. Note that you can control the optimizations to be performed with the optimization settings.

Optimize more/Build slower

When you enable this option, the compiler's frontend does not split the MIL stream in separate modules, but feeds it directly to the compiler's backend, allowing the code compaction to be performed application wide.

Related information

Section 4.1, *Compilation Process*

Control program option **--mil-link** / **--mil-split**

C compiler option: --misrac

Menu entry

1. Select **C/C++ Compiler » MISRA-C**.

2. Make a selection from the **MISRA-C checking** list.

3. If you selected **Custom**, expand the **Custom 2004** or **Custom 1998** entry and enable one or more individual rules.

Command line syntax

--misrac={**all** | *nr*[-*nr*]},...

Description

With this option you specify to the compiler which MISRA-C rules must be checked. With the option **--misrac=all** the compiler checks for all supported MISRA-C rules.

Example

```
c166 --misrac=9-13 test.c
```

The compiler generates an error for each MISRA-C rule 9, 10, 11, 12 or 13 violation in file `test.c`.

Related information

Section 4.8.2, *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

Menu entry

1. Select **C/C++ Compiler » MISRA-C**.

2. Make a selection from the **MISRA-C checking** list.

3. Enable one or both options **Warnings instead of errors for required rules** and **Warnings instead of errors for advisory rules**.

Command line syntax

```
--misrac-advisory-warnings
```

```
--misrac-required-warnings
```

Description

Normally, if an advisory rule or required rule is violated, the compiler generates an error. As a consequence, no output file is generated. With this option, the compiler generates a warning instead of an error.

Related information

Section 4.8.2, *C Code Checking: MISRA-C*

C compiler option **--misrac**

Linker option **--misrac-report**

C compiler option: --misrac-version

Menu entry

1.   Select **C/C++ Compiler » MISRA-C**.

2.   Select the **MISRA-C version**: **2004** or **1998**.

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 4.8.2, *C Code Checking: MISRA-C*

C compiler option **--misrac**

C compiler option: --model (-M)

Menu entry

1. Select **C/C++ Compiler » Memory Model**.

2. In the Default data box, select a memory model.

Command line syntax

--model={**near**|**far**|**shuge**|**huge**}

-M{**n**|**f**|**s**|**h**}

Default: **--model=near**

Description

By default, the C166 compiler uses the near memory model. With this memory model the most efficient code is generated. You can specify the option **--model** to specify another memory model.

The table below illustrates the meaning of each memory model:

Model	Memory type	Location	Pointer size	Pointer arithmetic
near	__near	Near data pages defined at link time	16 bit	16 bit
far	__far	Anywhere in memory	32 bit	14 bit
segmented huge	__shuge	Anywhere in memory	32 bit	16 bit
huge	__huge	Anywhere in memory	32 bit	32 bit

The value of the predefined preprocessor symbol __MODEL__ represents the memory model selected with this option. This can be very helpful in making conditional C code in one source module, used for different applications in different memory models. The value of __MODEL__ is:

near model	'n'
far model	'f'
segmented huge model	's'
huge model	'h'

Example

To compile the file `test.c` for the far memory model:

```
c166 --model=far test.c
```

Related information

Section 1.3.2, *Memory Models*

C compiler option: --near-functions

Menu entry

1. Select **C/C++ Compiler » Memory Model**.

2. Enable the option **Make unqualified functions near**.

Command line syntax

```
--near-functions
```

Description

With this option you tell the compiler to treat unqualified functions as $\_\_near$ functions instead of $\_\_huge$ functions. This function has the following effect:

*   CALLS instructions are changed into CALLA (no change in code size or performance)

*   JMPS instructions are changed into JMPA (no change in code size or performance)

*   return addresses on the stack will be 2 bytes shorter (save system stack)

Related information

-

C compiler option: --near-threshold

Menu entry

1. Select **C/C++ Compiler » Allocation**

2. In the **Threshold for putting data in __near** field, enter a value in bytes.

Command line syntax

--near-threshold=*threshold*

Default: **--near-threshold=0**

Description

With this option the compiler allocates unqualified objects that are smaller than or equal to the specified *threshold* in the __near memory space automatically. The *threshold* must be specified in bytes. Objects that are qualified const or volatile, external objects or objects that are absolute (__at()) are not moved.

By default the *threshold* is 0 (off), which means that all data is allocated in the default memory space.

You cannot use this option in the near (**--model=near**) memory model. This option has no effect when you use MIL linking/splitting (**--mil**/**--mil-split**). You can use the automatic near data allocation instead (**--automatic-near**).

Example

To put all data objects with a size of 4 bytes or smaller in __near memory:

```
c166 --model=far --near-threshold=4 test.c
```

Example part of test.c:

```
extern int x;   //will be put in default far memory
int y;          //will be put in near memory
```

Related information

C compiler option **--model** (Select memory model)

C compiler option **--mil** / **--mil-split**

C compiler option **--automatic-near** (Automatic near data allocation)

C compiler option: --no-clear / --no-clear-bit

Menu entry

1. Select **C/C++ Compiler » Allocation**.

2. Disable the option **Clear non-initialized global and static variables**.

3. Disable the option **Clear non-initialized global and static bit variables**.

Command line syntax

```
--no-clear
--no-clear-bit
```

Description

Normally 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.

Option **--no-clear-bit** is the same as **--no-clear**, except that it only applies to __bit variables.

Related information

Pragmas clear/noclear

Pragmas clear_bit/noclear_bit

C compiler option: --no-double (-F)

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Treat double as float**.

Command line syntax

```
--no-double
```

```
-F
```

Description

With this option you tell the compiler to treat variables of the type `double` as `float`. Because the float type takes less space, execution speed increases and code size decreases, both at the cost of less precision.

Example

```
c166 --no-double test.c
```

The file `test.c` is compiled where variables of the type double are treated as float.

Related information

-

C compiler option: --no-frame

Menu entry

1. Select **C/C++ Compiler » Code Generation**.

2. Disable the option **Generate frame for interrupt functions**.

Command line syntax

`--no-frame`

Description

With this option you tell the compiler not to generate an interrupt frame (saving/restoring registers) for interrupt handlers. In this case you will have to specify your own interrupt frame.

Related information

Section 1.12.4, *Interrupt Functions*

C compiler option: --no-savemac

Menu entry

1. Select **C/C++ Compiler » Code Generation**.

2. Disable the option **Save MAC registers in function prologue**.

Command line syntax

`--no-savemac`

Description

With this option the compiler will not save the MAC (multiply-accumulate) registers in the function prologue. This will make your code smaller and faster. This option also applies to `__interrupt()` functions.

Use this option only if you are sure that a function is never executed while the MAC unit is in use.

Related information

Pragmas `savemac/nosavemac`

C compiler option **--mac** (Automatic MAC code generation)

C compiler option: --no-stdinc

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Add the option **--no-stdinc** to the **Additional options** field.

Command line syntax

```
--no-stdinc
```

Description

With this option you tell the compiler not to look in the default `include` directory relative to the installation directory, when searching for include files. This way the compiler only searches in the include file search paths you specified.

Related information

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

Section 4.4, *How the Compiler Searches Include Files*

C compiler option: --no-tasking-sfr

Menu entry

1.   Select **C/C++ Compiler » Preprocessing**.

2.   Disable the option **Automatic inclusion of '.sfr' file**.

Command line syntax

```
--no-tasking-sfr
```

Description

Normally, the compiler includes a special function register (SFR) file before compiling. The compiler automatically selects the SFR file belonging to the target you selected on the **Processor** page (C compiler option **--cpu**).

With this option the compiler does not include the register file `regcpu.sfr` as based on the selected target processor.

Use this option if you want to use your own set of SFR files.

Related information

C compiler option **--cpu** (Select processor)

Section 1.3.5, *Accessing Hardware from C*

C compiler option: --no-warnings (-w)

Menu entry

1. Select **C/C++ Compiler » Diagnostics**.

    The Suppress C compiler warnings box shows the warnings that are currently suppressed.

2. To suppress a warning, click on the **Add** button in the **Suppress warnings** box.

3. Enter the numbers, separated by commas or as a range, of the warnings you want to suppress (for example `537,538`). Or you can use the **Add** button multiple times.

4. To suppress all warnings, enable the option **Suppress all warnings**.

> Use the **Edit** and **Delete** button to change a warning number or to remove a number from the list.

Command line syntax

`--no-warnings``[=`*number*`[-`*number*`],...]`

`-w``[`*number*`[-`*number*`],...]`

Description

With this option you can suppresses all warning messages or specific warning messages.

On the command line this option works as follows:

- If you do not specify this option, all warnings are reported.

- If you specify this option but without numbers, all warnings are suppressed.

- If you specify this option with a number or a range, only the specified warnings are suppressed. You can specify the option **--no-warnings=**number multiple times.

Example

To suppress warnings 537 and 538, enter:

```
c166 test.c --no-warnings=537,538
```

Related information

C compiler option **--warnings-as-errors** (Treat warnings as errors)

Pragma `warning`

C compiler option: --optimize (-O)

Menu entry

1. Select **C/C++ Compiler » Optimization**.

2. Select an optimization level in the **Optimization level** box.

Command line syntax

--optimize[**=***flags*]

-O*flags*

You can set the following flags:

+/-coalesce	**a/A**	Coalescer: remove unnecessary moves
+/-ipro	**b/B**	Interprocedural register optimizations
+/-cse	**c/C**	Common subexpression elimination
+/-expression	**e/E**	Expression simplification
+/-flow	**f/F**	Control flow simplification
+/-glo	**g/G**	Generic assembly code optimizations
+/-inline	**i/I**	Automatic function inlining
+/-schedule	**k/K**	Instruction scheduler
+/-loop	**l/L**	Loop transformations
+/-forward	**o/O**	Forward store
+/-propagate	**p/P**	Constant propagation
+/-compact	**r/R**	Code compaction (reverse inlining)
+/-subscript	**s/S**	Subscript strength reduction
+/-peephole	**y/Y**	Peephole optimizations
+/-predict		Branch prediction

Use the following options for predefined sets of flags:

--optimize=0	**-O0**	No optimization
		Alias for **-OaBCDEFGIKLOPRSY,-predict**

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 **-OabcefgIKLOPRSy,+predict**

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 **-Oabcefglkloprsy,+predict**

Enables more optimizations to reduce code size and/or execution time. This is the default optimization level.

--optimize=3	**-O3**	Optimize most
		Alias for **-Oabcefgikloprsy,+predict**

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:

```
c166 test.c

c166 --optimize=2 test.c
c166 -O2 test.c

c166 --optimize test.c
c166 -O test.c

c166 -OabcefgIkloprsy test.c
c166 --optimize=+coalesce,+ipro,+cse,+expression,+flow,+glo,
      -inline,+schedule,+loop,+forward,+propagate,+compact,
      +subscript,+peephole test.c
```

Related information

C compiler option **--tradeoff** (Trade off between speed and size)

Pragma `optimize/endoptimize`

Section 4.6, *Compiler Optimizations*

C compiler option: --option-file (-f)

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Add the option **--option-file** to the **Additional options** field.

   Be aware that the options in the option file are added to the C compiler options you have set in the other pages. Only in extraordinary cases you may want to use them in combination.

Command line syntax

```
--option-file=file,...
```

```
-f file,...
```

Description

This option is primarily intended for command line use. Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the compiler.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option **--option-file** multiple times.

Format of an option file

• Multiple arguments on one line in the option file are allowed.

• To include whitespace in an argument, surround the argument with single or double quotes.

• If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

```
"This has a single quote ' embedded"

'This has a double quote " embedded'

'This has a double quote " and a single quote '"' embedded"
```

• When a text line reaches its length limit, use a **** to continue the line. Whitespace between quotes is preserved.

```
"This is a continuation \
line"

        -> "This is a continuation line"
```

• It is possible to nest command line files up to 25 levels.

Example

Suppose the file `myoptions` contains the following lines:

```
--debug-info
--define=DEMO=1
test.c
```

Specify the option file to the compiler:

```
c166 --option-file=myoptions
```

This is equivalent to the following command line:

```
c166 —-debug-info --define=DEMO=1 test.c
```

Related information

-

C compiler option: --output (-o)

Menu entry

Eclipse names the output file always after the C source file.

Command line syntax

--output=*file*

-o *file*

Description

With this option you can specify another filename for the output file of the compiler. Without this option the basename of the C source file is used with extension `.src`.

Example

To create the file `output.src` instead of `test.src`, enter:

```
c166 --output=output.src test.c
```

Related information

-

C compiler option: --preprocess (-E)

Menu entry

1. Select **C/C++ Compiler » Preprocessing**.

2. Enable the option **Store preprocessor output in <file>.pre**.

3. (Optional) Enable the option **Keep comments in preprocessor output**.

4. (Optional) Enable the option **Keep #line info in preprocessor output**.

Command line syntax

--preprocess[**=***flags*]

-E[*flags*]

You can set the following flags:

+/-comments	**c/C**	keep comments
+/-includes	**i/I**	generate a list of included source files
+/-list	**l/L**	generate a list of macro definitions
+/-make	**m/M**	generate dependencies for make
+/-noline	**p/P**	strip #line source position information

Default: **-ECILMP**

Description

With this option you tell the compiler to preprocess the C source. Under Eclipse the compiler sends the preprocessed output to the file *name*.pre (where *name* is the name of the C source file to compile). Eclipse also compiles the C source.

On the command line, the compiler sends the preprocessed file to stdout. To capture the information in a file, specify an output file with the option **--output**.

With **--preprocess=+comments** you tell the preprocessor to keep the comments from the C source file in the preprocessed output.

With **--preprocess=+includes** the compiler will generate a list of all included source files. The preprocessor output is discarded.

With **--preprocess=+list** the compiler will generate a list of all macro definitions. The preprocessor output is discarded.

With **--preprocess=+make** the compiler will generate dependency lines that can be used in a Makefile. The preprocessor output is discarded. The default target name is the basename of the input file, with the extension .obj. With the option **--make-target** you can specify a target name which overrules the default target name.

With **--preprocess=+noline** you tell the preprocessor to strip the #line source position information (lines starting with #line). These lines are normally processed by the assembler and not needed in the preprocessed output. When you leave these lines out, the output is easier to read.

Example

```
c166 --preprocess=+comments,-make,-noline test.c --output=test.pre
```

The compiler preprocesses the file test.c and sends the output to the file test.pre. Comments are included but no dependencies are generated and the line source position information is not stripped from the output file.

Related information

C compiler option **--dep-file** (Generate dependencies in a file)

C compiler option **--make-target** (Specify target name for **-Em** output)

C compiler option: --profile (-p)

Menu entry

1. Select **C/C++ Compiler » Debugging**.

2. Enable or disable **Static profiling**.

3. Enable or disable one or more of the following **Generate profiling information** options (dynamic profiling):

   - **for block counters** (not in combination with Call graph or Function timers)

   - **to build a call graph** (not in combination with Block counters)

   - **for function counters**

   - **for function timers** (not in combination with Block counters/Function counters)

> Note that the more detailed information you request, the larger the overhead in terms of execution time, code size and heap space needed. The option **--debug** does not affect profiling, execution time or code size.

Command line syntax

--profile[**=**$flag$,...]

-p[$flags$]

Use the following option for a predefined set of flags:

--profile=g	**-pg**	Profiling with call graph and function timers. Alias for: **-pBcFSt**

You can set the following flags:

+/-block	**b/B**	block counters
+/-callgraph	**c/C**	call graph
+/-function	**f/F**	function counters
+/-static	**s/S**	static profile generation
+/-time	**t/T**	function timers

Default (without flags): **-pBCfST**

Description

Profiling is the process of collecting statistical data about a running application. With these data you can analyze which functions are called, how often they are called and what their execution time is.

Several methods of profiling exist. One method is *code instrumentation* which adds code to your application that takes care of the profiling process when the application is executed. Another method is *static profiling*.

For an extensive description of profiling refer to Chapter 6, *Profiling*.

You can obtain the following profiling data (see flags above):

Block counters (not in combination with Call graph or Function timers)

This will instrument the code to perform basic block counting. As the program runs, it counts the number of executions of each branch in an if statement, each iteration of a for loop, and so on. Note that though you can combine Block counters with Function counters, this has no effect because Function counters is only a subset of Block counters.

Call graph (not in combination with Block counters)

This will instrument the code to reconstruct the run-time call graph. As the program runs it associates the caller with the gathered profiling data.

Function counters

This will instrument the code to perform function call counting. This is a subset of the basic Block counters.

Function timers (not in combination with Block counters/Function counters)

This will instrument the code to measure the time spent in a function. This includes the time spent in all sub functions (callees).

Static profiling

With this option you do not need to run the application to get profiling results. The compiler generates profiling information at compile time, without adding extra code to your application.

If you use one or more profiling options that use code instrumentation, you must link the corresponding libraries too! Refer to Section 8.3, *Linking with Libraries*, for an overview of the (profiling) libraries. In Eclipse the correct libraries are linked automatically.

Example

To generate block count information for the module `test.c` during execution, compile as follows:

```
c166 --profile=+block test.c
```

In this case you must link the library `c166pbn.lib`.

Related information

Chapter 6, *Profiling*

C compiler option: --ramdata

Menu entry

-

Command line syntax

`--ramdata`

Description

With this option you tell the compiler to allocate non-automatic constant data in both ROM and RAM. At startup RAM is initialized from ROM. This option affects `const` variables, string literals, initializers and constants that are allocated in memory.

Related information

Pragmas `ramdata/noramdata`

Section 1.8.1, *Initialized Variables*

C compiler option: --rename-sections (-R)

Menu entry

1. Select **C/C++ Compiler » Allocation**

   The Rename sections box shows the sections that are currently renamed.

2. To rename a section, click on the **Add** button in the **Rename sections** box.

3. Type the rename rule in the format *type=format* or *format* (for example, `near={module}_{attrib}`)

Use the **Edit** and **Delete** button to change a section renaming or to remove an entry from the list.

Command line syntax

--rename-sections=[*type***=**]*format_string*[**,**[*type***=**]*format_string*]...

-R[*type***=**]*format_string*[**,**[*type***=**]*format_string*]...

Default section name: {type}_{name}

Description

In case a module must be loaded at a fixed address, or a data section needs a special place in memory, you can use this option to generate a different section name. You can then use this unique section name in the linker script file for locating.

With the memory *type* you select which sections are renamed. The matching sections will get the specified *format_string* for the 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
{type}   section type

Instead of this option you can also uses the pragmas `section/endsection` in the C source.

Example

To rename sections of memory type `near` to `_c166_test_`*variable_name*:

`c166 --rename-sections=near=_c166_{module}_{name} test.c`

Related information

See assembler directive `.SECTION` for a list of section types and attributes.

Pragmas `section/endsection`

Section 1.15, *Section Naming*

C compiler option: --romdata

Menu entry

-

Command line syntax

`--romdata`

Description

With this option you tell the compiler to allocate all non-automatic non-constant variables in ROM only. By default, the variables are allocated in RAM and initialized from ROM at startup.

Related information

Pragmas `romdata/noromdata`

Section 1.8.1, *Initialized Variables*

C compiler option: --runtime (-r)

Menu entry

1. Select **C/C++ Compiler » Debugging**.

2. Enable or disable one or more of the following run-time error checking options:

    - Generate code for bounds checking

    - Generate code to detect unhandled case in a switch

    - Generate code for malloc consistency checks

Command line syntax

--runtime[**=***flag*,...]

-r[*flags*]

You can set the following flags:

+/-bounds	**b/B**	bounds checking
+/-case	**c/C**	report unhandled case in a switch
+/-malloc	**m/M**	malloc consistency checks

Default (without flags): **-rbcm**

Description

This option controls a number of run-time checks to detect errors during program execution. Some of these checks require additional code to be inserted in the generated code, and may therefore slow down the program execution. The following checks are available:

Bounds checking

Every pointer update and dereference will be checked to detect out-of-bounds accesses, null pointers and uninitialized automatic pointer variables. This check will increase the code size and slow down the program considerably. In addition, some heap memory is allocated to store the bounds information. You may enable bounds checking for individual modules or even parts of modules only (see `#pragma runtime`).

Report unhandled case in a switch

Report an unhandled case value in a switch without a default part. This check will add one function call to every switch without a default part, but it will have little impact on the execution speed.

Malloc consistency checks

This option enables the use of wrappers around the functions malloc/realloc/free that will check for common dynamic memory allocation errors like:

- buffer overflow

- write to freed memory

- multiple calls to free

- passing invalid pointer to free

Enabling this check will extract some additional code from the library, but it will not enlarge your application code. The dynamic memory usage will increase by a couple of bytes per allocation.

Related information

Pragma `runtime`

C compiler option: --signed-bitfields

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Treat "int" bit-fields as signed**.

Command line syntax

```
--signed-bitfields
```

Description

For bit-fields it depends on the implementation whether a plain `int` is treated as `signed int` or `unsigned int`. By default an `int` bit-field is treated as `unsigned int`. This offers the best performance. With this option you tell the compiler to treat `int` bit-fields as `signed int`. In this case, you can still add the keyword `unsigned` to treat a particular `int` bit-field as `unsigned`.

Related information

C++ compiler option **--signed-bitfields**

Section 1.1, *Data Types*

C compiler option: --silicon-bug

Menu entry

1. Expand **C/C++ Build** and select **Processor**.

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

   *The **CPU problem bypasses and checks** box shows the available workarounds/checks available for the selected processor.*

3. (Optional) Select **Show all CPU problem bypasses and checks**.

4. Click **Select All** or select one or more individual options.

Command line syntax

```
--silicon-bug[=bug,...]
```

Description

With this option you specify for which hardware problems the compiler should generate workarounds. Please refer to Chapter 18, *CPU Problem Bypasses and Checks* for the numbers and descriptions. Silicon bug numbers are specified as a comma separated list. When this option is used without arguments, all silicon bug workarounds are enabled.

Example

To enable workarounds for problems CPU.11 and CPU.16, enter:

```
c166 --silicon-bug=4,5 test.c
```

Related information

Chapter 18, *CPU Problem Bypasses and Checks*

Assembler option **--silicon-bug**

C compiler option: --source (-s)

Menu entry

1.   Select **C/C++ Compiler » Miscellaneous**.

2.   Enable the option **Merge C source code with generated assembly**.

Command line syntax

`--source`

`-s`

Description

With this option you tell the compiler to merge C source code with generated assembly code in the output file. The C source lines are included as comments.

Related information

Pragmas `source/nosource`

C compiler option: --stack-address-conversion

Menu entry

1. Select **C/C++ Compiler » Code Generation**.

2. In the **Stack address conversion** field, select a mode.

Command line syntax

`--stack-address-conversion=`*mode*

You can specify the following modes:

dynamic	**d**	Determine DPP register dynamically
fixed-dpp	**b/B**	Use stack symbol's DPP register
static	**s**	Use stack symbol

Default: **static**

Description

With this option you can control how stack addresses are converted to `__far` and `__(s)huge` addresses. This can be useful for task switching.

static

Use the linker generated stack symbol to refer to the stack address. This is the default.

fixed-dpp

Use the DPP register that refers to the linker generated stack symbol to determine the stack address.

dynamic

Determine the used DPP register dynamically. The highest two bits of the near address determine the DPP register.

Related information

Pragma `stack_address_conversion`

C compiler option: --stdout (-n)

Menu entry

-

Command line syntax

--stdout

-n

Description

With this option you tell the compiler to send the output to stdout (usually your screen). No files are created. This option is for example useful to quickly inspect the output or to redirect the output to other tools.

Related information

-

C compiler option: --string-literal-memory

Menu entry

1. Select **C/C++ Compiler » Allocation**.

2. Select a **Memory space for string literals**.

Command line syntax

--string-literal-memory=*space*

Default: **--string-literal-memory=model**

Description

With this option you can control the allocation of string literals. The *space* must be one of __near, __far, __shuge, __huge or model.

When the *space* differs from the default memory model space, pointers to string literals may need qualification. Also, C library functions accepting a default pointer to char/wchar_t may need to be recompiled with a different name/prototype.

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 __near memory:

```
c166 --model=far --string-literal-memory=__near test.c
```

Related information

Pragma string_literal_memory

C compiler option **--model** (Select memory model)

Section 1.9, *Strings*

C compiler option: --tradeoff (-t)

Menu entry

1. Select **C/C++ Compiler » Optimization**.

2. Select a trade-off level in the **Trade-off between speed and size** box.

Command line syntax

`--tradeoff={0|1|2|3|4}`

`-t{0|1|2|3|4}`

Default: `--tradeoff=4`

Description

If the compiler uses certain optimizations (option **--optimize**), you can use this option to specify whether the used optimizations should optimize for more speed (regardless of code size) or for smaller code size (regardless of speed).

By default the compiler optimizes for code size (**--tradeoff=4**).

> If you have not specified the option **--optimize**, the compiler uses the default *Optimize more* optimization. In this case it is still useful to specify a trade-off level.

Example

To set the trade-off level for the used optimizations:

```
c166 --tradeoff=2 test.c
```

The compiler uses the default *Optimize more* optimization level and balances speed and size while optimizing.

Related information

C compiler option **--optimize** (Specify optimization level)

Section 4.6.3, *Optimize for Size or Speed*

C compiler option: --uchar (-u)

Menu entry

1.   Select **C/C++ Compiler » Language**.

2.   Enable the option **Treat "char" variables as unsigned**.

Command line syntax

`--uchar`

`-u`

Description

By default `char` is the same as specifying `signed char`. With this option `char` is the same as `unsigned char`.

Related information

Section 1.1, *Data Types*

C compiler option: --undefine (-U)

Menu entry

1. Select **C/C++ Compiler » Preprocessing**

    The Defined symbols box shows the symbols that are currently defined.

2. To remove a defined symbol, select the symbol in the **Defined symbols** box and click on the **Delete** button.

Command line syntax

--undefine=*macro_name*

-U*macro_name*

Description

With this option you can undefine an earlier defined macro as with #undef. This option is for example useful to undefine predefined macros.

The following predefined ISO C standard macros cannot be undefined:

__FILE__	current source filename
__LINE__	current source line number (int type)
__TIME__	hh:mm:ss
__DATE__	Mmm dd yyyy
__STDC__	level of ANSI standard

Example

To undefine the predefined macro __TASKING__:

c166 --undefine=__TASKING__ test.c

Related information

C compiler option **--define** (Define preprocessor macro)

Section 1.7, *Predefined Preprocessor Macros*

C compiler option: --user-stack

Menu entry

1.   Select **C/C++ Compiler » Memory Model**.

2.   Enable the option **Use user stack for return addresses**.

Command line syntax

`--user-stack`

Description

With this option the function return address is stored on the user stack instead of on the system stack. Also the user stack run-time libraries are used and the user stack calling convention is used to call the run-time library functions.

Related information

Section 1.12, *Functions*

C compiler option: --version (-V)

Menu entry

-

Command line syntax

--version

-V

Description

Display version information. The compiler ignores all other options or input files.

Example

```
c166 --version
```

The compiler does not compile any files but displays the following version information:

```
TASKING VX-toolset for C166: C compiler    vx.yrz Build nnn
Copyright 2004-year Altium BV               Serial# 00000000
```

Related information

-

C compiler option: --warnings-as-errors

Menu entry

1.   Select **Global Options**.

2.   Enable the option **Treat warnings as errors**.

Command line syntax

`--warnings-as-errors``[=`*number*`[-`*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`

12.3. C++ Compiler Options

This section lists all C++ compiler options.

Options in Eclipse versus options on the command line

Most command line options have an equivalent option in Eclipse but some options are only available on the command line. Eclipse invokes the C++ compiler via the control program. Therefore, it uses the syntax of the control program to pass options and files to the C++ compiler. If there is no equivalent option in Eclipse, you can specify a command line option in Eclipse as follows:

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

    The Properties dialog appears.

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

    In the right pane the Settings appear.

3. On the Tool Settings tab, select **C/C++ Compiler » Miscellaneous**.

4. In the **Additional options** field, enter one or more command line options.

    *Because Eclipse uses the control program, you have to precede the option with **-Wcp** to pass the option via the control program directly to the C++ compiler.*

Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (**-**) character, long option names always begin with two minus (**--**) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

If an option requires an argument, the argument may be separated from the keyword by white space, or the keyword may be immediately followed by **=***option*. When the second form is used there may not be any white space on either side of the equal sign.

Options can have flags or suboptions. To switch a flag 'on', use a lowercase letter or a **+***longflag*. To switch a flag off, use an uppercase letter or a **-***longflag*. Separate *longflags* with commas. The following two invocations are equivalent:

```
cp166 -Ecp test.cc
cp166 --preprocess=+comments,+noline test.cc
```

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

The priority of the options is left-to-right: when two options conflict, the first (most left) one takes effect. The **-D** and **-U** options are not considered conflicting options, so they are processed left-to-right for each source file. You can overrule the default output file name with the **--output-file** option.

C++ compiler option: --alternative-sfr-file

Menu entry

1. Select **C/C++ Compiler » Preprocessing**.

2. Enable the option **Use alternative SFR file format**.

Command line syntax

```
--alternative-sfr-file
```

Description

With this option the C++ compiler includes the alternative SFR file without alias definitions for SFRs and SFR bit-fields. This alternative SFR file is named `reg`*cpu*`.asfr` and is located in the product's `include/sfr` directory.

Use this option to speed up your compilation (smaller SFR file) and have less namespace pollution.

Related information

Section 1.3.5, *Accessing Hardware from C*

C++ compiler option: --alternative-tokens

Menu entry

-

Command line syntax

--alternative-tokens

Description

Enable recognition of alternative tokens. This controls recognition of the digraph tokens in C++, and controls recognition of the operator keywords (e.g., `not`, `and`, `bitand`, etc.).

Example

To enable operator keywords (e.g., "not", "and") and digraphs, enter:

```
cp166 --alternative-tokens test.cc
```

Related information

-

C++ compiler option: --anachronisms

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **C++ anachronisms**.

Command line syntax

```
--anachronisms
```

Description

Enable C++ anachronisms. This option also enables **--nonconst-ref-anachronism**. But you can turn this off individually with option **--no-nonconst-ref-anachronism**.

Related information

C++ compiler option **--nonconst-ref-anachronism** (Nonconst reference anachronism)

Section 2.2.3, *Anachronisms Accepted*

C++ compiler option: --base-assign-op-is-default

Menu entry

-

Command line syntax

`--base-assign-op-is-default`

Description

Enable the anachronism of accepting a copy assignment operator that has an input parameter that is a reference to a base class as a default copy assignment operator for the derived class.

Related information

-

C++ compiler option: --bita-struct-threshold

Menu entry

1. Select **C/C++ Compiler » Allocation**

2. In the **Threshold for putting structs in __bita** field, enter a value in bytes.

Command line syntax

`--bita-struct-threshold=`*threshold*

Default: `--bita-struct-threshold=0`

Description

With this option the C++ compiler allocates unqualified structures that are smaller than or equal to the specified *threshold* and have a bit-field of length one in the bit-addressable (`__bita`) memory space automatically. The *threshold* must be specified in bytes. Objects that are qualified `const` or `volatile` or objects that are absolute are not moved.

By default the *threshold* is 0 (off), which means that no objects are moved.

Example

To put all unqualified structures with a size of 4 bytes or smaller and a bit-field of length one into the `__bita` section:

```
cp166 --bita-struct-threshold=4 test.cc
```

Related information

-

C++ compiler option: --building-runtime

Menu entry

-

Command line syntax

`--building-runtime`

Description

Special option for building the C++ run-time library. Used to indicate that the C++ run-time library is being compiled. This causes additional macros to be predefined that are used to pass configuration information from the C++ compiler to the run-time.

Related information

-

C++ compiler option: --c++0x

Menu entry

-

Command line syntax

`--c++0x`

Description

Enable the C++ extensions that are defined by the latest C++ working paper.

Related information

-

C++ compiler option: --check

Menu entry

-

Command line syntax

`--check`

Description

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

The C++ compiler reports any warnings and/or errors.

This option is available on the command line only.

Related information

C compiler option **--check** (Check syntax)

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

C++ compiler option: --context-limit

Menu entry

-

Command line syntax

`--context-limit=`*number*

Default: `--context-limit=10`

Description

Set the context limit to *number.* The context limit is the maximum number of template instantiation context entries to be displayed as part of a diagnostic message. If the number of context entries exceeds the limit, the first and last *N* context entries are displayed, where *N* is half of the context limit. A value of zero is used to indicate that there is no limit.

Example

To set the context limit to 5, enter:

```
cp166 --context-limit=5 test.cc
```

Related information

-

C++ compiler option: --core

Menu entry

1. Expand **C/C++ Build** and select **Processor**.

2. From the **Processor selection** list, expand **Custom** and select a core.

Command line syntax

--core=_core_

You can specify the following _core_ arguments:

c16x	C16x instruction set
st10	ST10 instruction set
st10mac	ST10 with MAC co-processor support
c166sv1	C166S v1.0 support
xc16x	XC16x/XE16xx/XC2xxx instruction set
super10	Super10 instruction set
super10m345	Enhanced Super10M345 instruction set

Default: derived from **--cpu** if specified and known or else **--core=xc16x**

Description

With this option you specify the core architecture for a target processor for which you create your application. If **--cpu** is specified and the supplied CPU is known by the C++ compiler, the C++ compiler selects the correct core automatically.

The macro __CORE__ is set to the name of the _core_.

For more information see C++ compiler option **--cpu**.

Example

Specify a custom core:

```
cp166 --core=st10 test.cc
```

Related information

C++ compiler option **--cpu** (Select processor)

C++ compiler option: --cpu (-C)

Menu entry

1. Expand **C/C++ Build** and select **Processor**.

2. From the **Processor selection** list, select a processor or expand **Custom** and select a core.

Command line syntax

--cpu=*cpu*

-C*cpu*

Description

With this option you define the target processor for which you create your application.

Based on this option the C++ compiler always includes the special function register file reg*cpu*.sfr, unless you specify option **--no-tasking-sfr**.

If you select a target from the list, the core is known. If you specify a Custom processor, you need to select the core that matches the core of your custom processor (option **--core**). The C++ compiler knows all CPUs with core c16x, st10, st10mac and super10. If you specify a CPU that is not in that list, it is assumed to be an xc16x.

The following table show the relation between the two options:

--cpu=*cpu*	--core=*core*	*core*	Register file
no	no	c16x	c166, cp166: none as166: regc16x.sfr
no	yes	*core*	c166, cp166: none as166: reg*core*.sfr
yes	no	derived from *cpu* for core c16x, st10, st10mac and super10 If the *cpu* is unknown core xc16x is assumed	reg*cpu*.sfr
yes	yes	*core*	reg*cpu*.sfr

The macro __CPU__ is set to the name of the *cpu*.

The standard list of supported processors is defined in the file processors.xml. This file defines for each processor its full name (for example, XC2287-72F), the base CPU name (for example, xc2287), the core settings (for example, xc16x), the on-chip flash settings, the list of silicon bugs for that processor. Each processor also defines an option to supply to the linker for preprocessing the LSL file for the applicable on-chip memory definitions. The option is for example -DXC2287_72M.

Example

Specify an existing processor:

```
cp166 --cpu=c167cs40 test.cc
```

Specify a custom processor:

```
cp166 --cpu=custom --core=st10 test.cc
```

Related information

C++ compiler option **--core** (Select the core)

C++ compiler option **--no-tasking-sfr** (Do not include SFR file)

C++ compiler option: --create-pch

Menu entry

1. Select **C/C++ Compiler » Precompiled C++ Headers**.

2. Enter a filename in the **Create precompiled header file** field.

Command line syntax

--create-pch=*filename*

Description

If other conditions are satisfied, create a precompiled header file with the specified name. If **--pch** (automatic PCH mode) or **--use-pch** appears on the command line following this option, its effect is erased.

Example

To create a precompiled header file with the name test.pch, enter:

```
cp166 --create-pch=test.pch test.cc
```

Related information

C++ compiler option **--pch** (Automatic PCH mode)

C++ compiler option **--use-pch** (Use precompiled header file)

Section 2.10, *Precompiled Headers*

C++ compiler option: --define (-D)

Menu entry

1. Select **C/C++ Compiler » Preprocessing**.

    The Defined symbols box shows the symbols that are currently defined.

2. To define a new symbol, click on the **Add** button in the **Defined symbols** box.

3. Type the symbol definition (for example, `demo=1`)

Use the **Edit** and **Delete** button to change a macro definition or to remove a macro from the list.

Command line syntax

--define=`macro_name`[**(**`parm-list`**)**][**=**`macro_definition`]

-D`macro_name`**(**`parm-list`**)**][**=**`macro_definition`]

Description

With this option you can define a macro and specify it to the preprocessor. If you only specify a macro name (no macro definition), the macro expands as '1'.

Function-style macros can be defined by appending a macro parameter list to *macro_name*.

You can specify as many macros as you like. Simply use the **Add** button to add new macro definitions.

On the command line, you can use the option **--define** (**-D**) multiple times. If the command line exceeds the limit of the operating system, you can define the macros in an option file which you then must specify to the C++ compiler with the option **--option-file** (**-f**) *file*.

The definition can be tested by the preprocessor with `#if`, `#ifdef` and `#ifndef`, for conditional compilations.

Example

Consider the following program with conditional code to compile a demo program and a real program:

```
void main( void )
{
#if DEMO
    demo_func();   /* compile for the demo program */
#else
    real_func();   /* compile for the real program */
#endif
}
```

You can now use a macro definition to set the DEMO flag:

```
cp166 --define=DEMO test.cc
cp166 --define=DEMO=1 test.cc
```

Note that both invocations have the same effect.

The next example shows how to define a macro with arguments. Note that the macro name and definition are placed between double quotes because otherwise the spaces would indicate a new option.

```
cp166 --define="MAX(A,B)=((A) > (B) ? (A) : (B))" test.cc
```

Related information

C++ compiler option **--undefine** (Remove preprocessor macro)

C++ compiler option **--option-file** (Specify an option file)

C++ compiler option: --dep-file

Menu entry

-

Command line syntax

--dep-file[**=***file*]

Description

With this option you tell the C++ compiler to generate dependency lines that can be used in a Makefile. In contrast to the option **--preprocess=+make**, the dependency information will be generated in addition to the normal output file.

By default, the information is written to a file with extension .d (one for every input file). When you specify a filename, all dependencies will be combined in the specified file.

Example

```
cp166 --dep-file=test.dep test.cc
```

The C++ compiler compiles the file test.cc, which results in the output file test.ic, and generates dependency lines in the file test.dep.

Related information

C++ compiler option **--preprocess=+make** (Generate dependencies for make)

C++ compiler option: --dollar

Menu entry

-

Command line syntax

`--dollar`

Default format: No dollar signs are allowed in identifiers.

Description

Accept dollar signs in identifiers. Names like A$VAR are allowed.

Related information

-

C++ compiler option: --embedded-c++

Menu entry

1.   Select **C/C++ Compiler » Language**.

2.   Enable the option **Comply to embedded C++ subset**.

Command line syntax

`--embedded-c++`

Description

The "Embedded C++" subset does not support templates, exceptions, namespaces, new-style casts, RTTI, multiple inheritance, virtual base classes, and the `mutable` keyword. Select this option when you want the C++ compiler to give an error when you use any of them in your C++ source.

Related information

-

C++ compiler option: --error-file

Menu entry

-

Command line syntax

--error-file[**=***file*]

Description

With this option the C++ compiler redirects error messages to a file. If you do not specify a filename, the error file will be named after the input file with extension `.ecp`.

Example

To write errors to `errors.ecp` instead of `stderr`, enter:

```
cp166 --error-file=errors.ecp test.cc
```

Related information

-

C++ compiler option: --error-limit (-e)

Menu entry

-

Command line syntax

`--error-limit=``number`

`-e``number`

Default: **`--error-limit=100`**

Description

Set the error limit to *number*. The C++ compiler will abandon compilation after this number of errors (remarks and warnings are not counted). By default, the limit is 100.

Example

When you want compilation to stop when 10 errors occurred, enter:

```
cp166 --error-limit=10 test.cc
```

Related information

-

C++ compiler option: --exceptions (-x)

Menu entry

1.   Select **C/C++ Compiler » Language**.

2.   Enable the option **Support for C++ exception handling**.

Command line syntax

```
--exceptions
```

```
-x
```

Description

With this option you enable support for exception handling in the C++ compiler.

The macro __EXCEPTIONS is defined when exception handling support is enabled.

Related information

-

C++ compiler option: --exported-template-file

Menu entry

-

Command line syntax

--exported-template-file=`file`

Description

This option specifies the name to be used for the exported template file used for processing of exported templates.

This option is supplied for use by the control program that invokes the C++ compiler and is not intended to be used by end-users.

Related information

-

C++ compiler option: --extended-variadic-macros

Menu entry

-

Command line syntax

`--extended-variadic-macros`

Default: macros with a variable number of arguments are not allowed.

Description

Allow macros with a variable number of arguments (implies **--variadic-macros**) and allow the naming of the variable argument list.

Related information

C++ compiler option **--variadic-macros** (Allow variadic macros)

C++ compiler option: --force-vtbl

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Enable the option **Force definition of virtual function tables (C++)**.

Command line syntax

```
--force-vtbl
```

Description

Force definition of virtual function tables in cases where the heuristic used by the C++ compiler to decide on definition of virtual function tables provides no guidance.

Related information

C++ compiler option **--suppress-vtbl** (Suppress definition of virtual function tables)

C++ compiler option: --friend-injection

Menu entry

-

Command line syntax

--friend-injection

Default: `friend` names are not injected.

Description

Controls whether the name of a class or function that is declared only in `friend` declarations is visible when using the normal lookup mechanisms. When `friend` names are injected, they are visible to such lookups. When `friend` names are not injected (as required by the standard), function names are visible only when using argument-dependent lookup, and class names are never visible.

Related information

C++ compiler option **--no-arg-dep-lookup** (Disable argument dependent lookup)

C++ compiler option: --g++

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Allow GNU C++ extensions**.

Command line syntax

`--g++`

Description

Enable GNU C++ compiler language extensions.

Related information

Section 2.3, *GNU Extensions*

C++ compiler option: --gnu-version

Menu entry

-

Command line syntax

`--gnu-version=`*version*

Default: 30300 (version 3.3.0)

Description

It depends on the GNU C++ compiler version if a particular GNU extension is supported or not. With this option you set the GNU C++ compiler version that should be emulated in GNU C++ mode. Version *x.y.z* of the GNU C++ compiler is represented by the value *x**10000+*y**100+*z*.

Example

To specify version 3.4.1 of the GNU C++ compiler, enter:

```
cp166 --g++ --gnu-version=30401 test.cc
```

Related information

Section 2.3, *GNU Extensions*

C++ compiler option: --guiding-decls

Menu entry

-

Command line syntax

`--guiding-decls`

Description

Enable recognition of "guiding declarations" of template functions. A guiding declaration is a function declaration that matches an instance of a function template but has no explicit definition (since its definition derives from the function template). For example:

```
template <class T> void f(T) { ... }
void f(int);
```

When regarded as a guiding declaration, `f(int)` is an instance of the template; otherwise, it is an independent function for which a definition must be supplied.

Related information

C++ compiler option **--old-specializations** (Old-style template specializations)

C++ compiler option: --help (-?)

Menu entry

-

Command line syntax

--help[**=***item*]

-?

You can specify the following arguments:

      options   Show extended option descriptions

Description

Displays an overview of all command line options. When you specify an argument you can list extended information such as a list of option descriptions.

Example

The following invocations all display a list of the available command line options:

```
cp166 -?
cp166 --help
cp166
```

The following invocation displays an extended list of the available options:

```
cp166 --help=options
```

Related information

-

C++ compiler option: --implicit-extern-c-type-conversion

Menu entry

-

Command line syntax

`--implicit-extern-c-type-conversion`

Description

Enable the implicit type conversion between pointers to `extern "C"` and `extern "C++"` function types.

Related information

-

C++ compiler option: --implicit-include

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Enable the option **Implicit inclusion of source files for finding templates**.

Command line syntax

```
--implicit-include
```

Description

Enable implicit inclusion of source files as a method of finding definitions of template entities to be instantiated.

Related information

C++ compiler option **--instantiate** (Instantiation mode)

Section 2.5, *Template Instantiation*

C++ compiler option: --incl-suffixes

Menu entry

-

Command line syntax

--incl-suffixes=_suffixes_

Default: no extension and `.stdh`.

Description

Specifies the list of suffixes to be used when searching for an include file whose name was specified without a suffix. If a null suffix is to be allowed, it must be included in the suffix list. *suffixes* is a colon-separated list of suffixes (e.g., "`::stdh`").

Example

To allow only the suffixes `.h` and `.stdh` as include file extensions, enter:

```
cp166 --incl-suffixes=h:stdh test.cc
```

Related information

C++ compiler option **--include-file** (Include file at the start of a compilation)

Section 5.2, *How the C++ Compiler Searches Include Files*

C++ compiler option: --include-directory (-I)

Menu entry

1. Select **C/C++ Compiler » Include Paths**.

   The Include paths box shows the directories that are added to the search path for include files.

2. To define a new directory for the search path, click on the **Add** button in the **Include paths** box.

3. Type or select a path.

Use the **Edit** and **Delete** button to change a path or to remove a path from the list.

Command line syntax

--include-directory=`path`,...

-I`path`,...

Description

Add *path* to the list of directories searched for `#include` files whose names do not have an absolute pathname. You can specify multiple directories separated by commas.

Example

To add the directory `/proj/include` to the include file search path, enter:

```
cp166 --include-directory=/proj/include test.cc
```

Related information

C++ compiler option **--include-file** (Include file at the start of a compilation)

C++ compiler option **--sys-include** (Add directory to system include file search path)

Section 5.2, *How the C++ Compiler Searches Include Files*

C++ compiler option: --include-file (-H)

Menu entry

1.   Select **C/C++ Compiler » Preprocessing**.

     The Pre-include files box shows the files that are currently included before the compilation starts.

2.   To define a new file, click on the **Add** button in the **Include files at start of compilation** box.

3.   Type the full path and file name or select a file.

Use the **Edit** and **Delete** button to change a file name or to remove a file from the list.

Command line syntax

--include-file=*file*

-H*file*

Description

Include the source code of the indicated file at the beginning of the compilation. This is the same as specifying #include "file" at the beginning of *each* of your C++ sources.

All files included with **--include-file** are processed after any of the files included with **--include-macros-file**.

The filename is searched for in the directories on the include search list.

Example

cp166 --include-file=extra.h test1.cc test2.cc

The file extra.h is included at the beginning of both test1.cc and test2.cc.

Related information

C++ compiler option **--include-directory** (Add directory to include file search path)

Section 5.2, *How the C++ Compiler Searches Include Files*

C++ compiler option: --include-macros-file

Menu entry

-

Command line syntax

`--include-macros-file=`*file*

Description

Include the macros of the indicated file at the beginning of the compilation. Only the preprocessing directives from the file are evaluated. All of the actual code is discarded. The effect of this option is that any macro definitions from the specified file will be in effect when the primary source file is compiled. All of the macro-only files are processed before any of the normal includes (**--include-file**). Within each group, the files are processed in the order in which they were specified.

Related information

C++ compiler option **--include-file** (Include file at the start of a compilation)

Section 5.2, *How the C++ Compiler Searches Include Files*

C++ compiler option: --init-priority

Menu entry

-

Command line syntax

--init-priority=ary*number*

Default: 0

Description

Normally, the C++ compiler assigns no priority to the global initialization functions and the exact order is determined by the linker. This option sets the default priority for global initialization functions. Default value is "0". You can also set the default priority with the `#pragma init_priority`.

Values from 1 to 100 are for internal use only and should not be used. Values 101 to 65535 are available for user code. A lower number means a higher priority.

Example

```
cp166 --init-priority=101 test.cc
```

Related information

-

C++ compiler option: --instantiate (-t)

Menu entry

1.   Select **C/C++ Compiler » Miscellaneous**.

2.   Select an instantiation mode in the **Instantiation mode of external template entities** box.

Command line syntax

--instantiate=*mode*

-t*mode*

You can specify the following modes:

> **used**
>
> **all**
>
> **local**

Default: **--instantiate=used**

Description

Control instantiation of external template entities. External template entities are external (that is, non-inline and non-static) template functions and template static data members. The instantiation mode determines the template entities for which code should be generated based on the template definition. Normally, when a file is compiled, template entities are instantiated wherever they are used (the linker will discard duplicate definitions). The overall instantiation mode can, however, be changed with this option. You can specify the following modes:

used	Instantiate those template entities that were used in the compilation. This will include all static data members for which there are template definitions. This is the default.
all	Instantiate all template entities declared or referenced in the compilation unit. For each fully instantiated template class, all of its member functions and static data members will be instantiated whether or not they were used. Non-member template functions will be instantiated even if the only reference was a declaration.
local	Similar to **--instantiate=used** except that the functions are given internal linkage. This is intended to provide a very simple mechanism for those getting started with templates. The compiler will instantiate the functions that are used in each compilation unit as local functions, and the program will link and run correctly (barring problems due to multiple copies of local static variables). However, one may end up with many copies of the instantiated functions, so this is not suitable for production use.

You cannot use **--instantiate=local** in conjunction with automatic template instantiation.

Related information

C++ compiler option **--no-auto-instantiation** (Disable automatic C++ instantiation)

Section 2.5, *Template Instantiation*

C++ compiler option: --integer-enumeration

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Treat enumerated types always as integer**.

Command line syntax

`--integer-enumeration`

Description

Normally the C++ compiler treats enumerated types as the smallest data type possible (`char` instead of `int`). This reduces code size. With this option the C++ compiler always treats enum-types as `int` as defined in the ISO C99 standard.

Related information

Section 1.1, *Data Types*

C++ compiler option: --io-streams

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Support for C++ I/O streams**.

Command line syntax

`--io-streams`

Description

As I/O streams require substantial resources they are disabled by default. Use this option to enable I/O streams support in the C++ library.

This option also enables exception handling.

Related information

-

C++ compiler option: --late-tiebreaker

Menu entry

-

Command line syntax

`--late-tiebreaker`

Default: early tiebreaker processing.

Description

Select the way that tie-breakers (e.g., cv-qualifier differences) apply in overload resolution. In "early" tie-breaker processing, the tie-breakers are considered at the same time as other measures of the goodness of the match of an argument value and the corresponding parameter type (this is the standard approach).

In "late" tie-breaker processing, tie-breakers are ignored during the initial comparison, and considered only if two functions are otherwise equally good on all arguments; the tie-breakers can then be used to choose one function over another.

Related information

-

C++ compiler option: --list-file (-L)

Menu entry

-

Command line syntax

--list-file=_file_

-L_file_

Default: -1

Description

Generate raw listing information in the *file*. This information is likely to be used to generate a formatted listing. The raw listing file contains raw source lines, information on transitions into and out of include files, and diagnostics generated by the C++ compiler.

Each line of the listing file begins with a key character that identifies the type of line, as follows:

N A normal line of source; the rest of the line is the text of the line.

X The expanded form of a normal line of source; the rest of the line is the text of the line. This line appears following the N line, and only if the line contains non-trivial modifications (comments are considered trivial modifications; macro expansions, line splices, and trigraphs are considered non-trivial modifications). Comments are replaced by a single space in the expanded-form line.

S A line of source skipped by an `#if` or the like; the rest of the line is text. Note that the `#else`, `#elif`, or `#endif` that ends a skip is marked with an N.

L An indication of a change in source position. The line has a format similar to the # line-identifying directive output by the C preprocessor, that is to say

    `L` *line_number* `"file-name"` [*key*]

where *key* is, **1** for entry into an include file, or **2** for exit from an include file, and omitted otherwise.

The first line in the raw listing file is always an L line identifying the primary input file. L lines are also output for #line directives (key is omitted). L lines indicate the source position of the following source line in the raw listing file.

R, **W**, **E**, or **C** An indication of a diagnostic (**R** for remark, **W** for warning, **E** for error, and **C** for catastrophic error). The line has the form:

```
S   "file-name" line_number column-number message-text
```

where *S* is R, W, E, or C, as explained above. Errors at the end of file indicate the last line of the primary source file and a column number of zero. Command line errors are catastrophes with an empty file name ("") and a line and column number of zero. Internal errors are catastrophes with position information as usual, and message-text beginning with (internal error). When a diagnostic displays a list (e.g., all the contending routines when there is ambiguity on an overloaded call), the initial diagnostic line is followed by one or more lines with the same overall format (code letter, file name, line number, column number, and message text), but in which the code letter is the lower case version of the code letter in the initial line. The source position in such lines is the same as that in the corresponding initial line.

Example

To write raw listing information to the file `test.lst`, enter:

```
cp166 --list-file=test.lst test.cc
```

Related information

-

C++ compiler option: --long-lifetime-temps

Menu entry

-

Command line syntax

`--long-lifetime-temps`

Description

Select the lifetime for temporaries: short means to end of full expression; long means to the earliest of end of scope, end of switch clause, or the next label. Short is the default.

Related information

-

C++ compiler option: --long-long

Menu entry

-

Command line syntax

`--long-long`

Description

Permit the use of `long long` in strict mode in dialects in which it is non-standard.

Related information

-

C++ compiler option: --make-target

Menu entry

-

Command line syntax

`--make-target=``name`

Description

With this option you can overrule the default target name in the make dependencies generated by the options **--preprocess=+make** (**-Em**) and **--dep-file**. The default target name is the basename of the input file, with extension `.obj`.

Example

```
cp166 --preprocess=+make --make-target=mytarget.obj test.cc
```

The compiler generates dependency lines with the default target name `mytarget.obj` instead of `test.obj`.

Related information

C++ compiler option **--preprocess=+make** (Generate dependencies for make)

C++ compiler option **--dep-file** (Generate dependencies in a file)

C++ compiler option: --model (-M)

Menu entry

1. Select **C/C++ Compiler » Memory Model**.

2. In the Default data box, select a memory model.

Command line syntax

--model={near|far|shuge|huge}

-M{n|f|s|h}

Default: **--model=near**

Description

Select the memory model for the C++ compiler.

Example

To compile the file test.cc for the far memory model:

```
cp166 --model=far test.cc
```

Related information

C compiler option **--model** (Select memory model)

Section 1.3.2, *Memory Models*

C++ compiler option: --multibyte-chars

Menu entry

-

Command line syntax

`--multibyte-chars`

Default: multibyte character sequences are not allowed.

Description

Enable processing for multibyte character sequences in comments, string literals, and character constants. Multibyte encodings are used for character sets like the Japanese SJIS.

Related information

-

C++ compiler option: --namespaces

Menu entry

-

Command line syntax

`--namespaces`

`--no-namespaces`

Default: namespaces are supported.

Description

When you used option **--embedded-c++** namespaces are disabled. With option **--namespaces** you can enable support for namespaces in this case.

The macro __NAMESPACES is defined when namespace support is enabled.

Related information

C++ compiler option **--embedded-c++** (Embedded C++ compliancy tests)

C++ compiler option **--using-std** (Implicit use of the std namespace)

Section 2.4, *Namespace Support*

C++ compiler option: --near-functions

Menu entry

-

Command line syntax

`--near-functions`

Description

With this option you tell the C++ compiler to treat unqualified functions as `__near` functions instead of `__huge` functions.

Related information

C compiler option **--near-functions**

C++ compiler option: --near-threshold

Menu entry

1.   Select **C/C++ Compiler » Allocation**

2.   In the **Threshold for putting data in __near** field, enter a value in bytes.

Command line syntax

--near-threshold=*threshold*

Default: **--near-threshold=0**

Description

With this option the C++ compiler allocates unqualified objects that are smaller than or equal to the specified *threshold* in the __near memory space automatically. The *threshold* must be specified in bytes. Objects that are qualified const or volatile, external objects or objects that are absolute (__at()) are not moved.

By default the *threshold* is 0 (off), which means that all data is allocated in the default memory space.

You cannot use this option in the near (**--model=near**) memory model.

Example

To put all data objects with a size of 4 bytes or smaller in __near memory:

```
cp166 --model=far --near-threshold=4 test.cc
```

Example part of test.cc:

```
extern int x;  //will be put in default far memory
int y;         //will be put in near memory
```

Related information

C++ compiler option **--model** (Select memory model)

C compiler option **--model** (Select memory model)

C++ compiler option: --no-arg-dep-lookup

Menu entry

-

Command line syntax

`--no-arg-dep-lookup`

Default: argument dependent lookup of unqualified function names is performed.

Description

With this option you disable argument dependent lookup of unqualified function names.

Related information

-

C++ compiler option: --no-array-new-and-delete

Menu entry

-

Command line syntax

`--no-array-new-and-delete`

Default: array new and delete are supported.

Description

Disable support for array new and delete.

The macro `__ARRAY_OPERATORS` is defined when array new and delete is enabled.

Related information

-

C++ compiler option: --no-auto-instantiation

Menu entry

-

Command line syntax

`--no-auto-instantiation`

Default: the C++ compiler automatically instantiates templates.

Description

With this option automatic instantiation of templates is disabled.

Related information

C++ compiler option **--instantiate** (Set instantiation mode)

Section 2.5, *Template Instantiation*

C++ compiler option: --no-bool

Menu entry

-

Command line syntax

--no-bool

Default: `bool` is recognized as a keyword.

Description

Disable recognition of the `bool` keyword.

The macro `_BOOL` is defined when `bool` is recognized as a keyword.

Related information

-

C++ compiler option: --no-class-name-injection

Menu entry

-

Command line syntax

`--no-class-name-injection`

Default: the name of a class is injected into the scope of the class (as required by the standard).

Description

Do not inject the name of a class into the scope of the class (as was true in earlier versions of the C++ language).

Related information

-

C++ compiler option: --no-const-string-literals

Menu entry

-

Command line syntax

`--no-const-string-literals`

Default: C++ string literals and wide string literals are `const` (as required by the standard).

Description

With this option C++ string literals and wide string literals are non-const (as was true in earlier versions of the C++ language).

Related information

-

C++ compiler option: --no-dep-name

Menu entry

-

Command line syntax

`--no-dep-name`

Default: dependent name processing is enabled.

Description

Disable dependent name processing; i.e., the special lookup of names used in templates as required by the C++ standard. This option implies the use of **--no-parse-templates**.

Related information

C++ compiler option **--no-parse-templates** (Disable parsing of nonclass templates)

C++ compiler option: --no-distinct-template-signatures

Menu entry

-

Command line syntax

`--no-distinct-template-signatures`

Description

Control whether the signatures for template functions can match those for non-template functions when the functions appear in different compilation units. By default a normal function cannot be used to satisfy the need for a template instance; e.g., a function "`void f(int)`" could not be used to satisfy the need for an instantiation of a template "`void f(T)`" with `T` set to `int`.

--no-distinct-template-signatures provides the older language behavior, under which a non-template function can match a template function. Also controls whether function templates may have template parameters that are not used in the function signature of the function template.

Related information

-

C++ compiler option: --no-double (-F)

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Treat double as float**.

Command line syntax

--no-double

-F

Description

With this option you tell the C++ compiler to treat variables of the type double as float. Because the float type takes less space, execution speed increases and code size decreases, both at the cost of less precision.

Example

cp166 --no-double test.cc

The file test.cc is compiled where variables of the type double are treated as float.

Related information

-

C++ compiler option: --no-enum-overloading

Menu entry

-

Command line syntax

`--no-enum-overloading`

Description

Disable support for using operator functions to overload built-in operations on enum-typed operands.

Related information

-

C++ compiler option: --no-explicit

Menu entry

-

Command line syntax

--no-explicit

Default: the `explicit` specifier is allowed.

Description

Disable support for the `explicit` specifier on constructor declarations.

Related information

-

C++ compiler option: --no-export

Menu entry

-

Command line syntax

`--no-export`

Default: exported templates (declared with the keyword `export`) are allowed.

Description

Disable recognition of exported templates. This option requires that dependent name processing be done, and cannot be used with implicit inclusion of template definitions.

Related information

Section 2.5.5, *Exported Templates*

C++ compiler option: --no-extern-inline

Menu entry

-

Command line syntax

--no-extern-inline

Default: `inline` functions are allowed to have external linkage.

Description

Disable support for `inline` functions with external linkage in C++. When `inline` functions are allowed to have external linkage (as required by the standard), then `extern` and `inline` are compatible specifiers on a non-member function declaration; the default linkage when `inline` appears alone is external (that is, `inline` means `extern inline` on non-member functions); and an `inline` member function takes on the linkage of its class (which is usually external). However, when `inline` functions have only internal linkage (using **--no-extern-inline**), then `extern` and `inline` are incompatible; the default linkage when `inline` appears alone is internal (that is, `inline` means `static inline` on non-member functions); and `inline` member functions have internal linkage no matter what the linkage of their class.

Related information

Section 2.7, *Extern Inline Functions*

C++ compiler option: --no-for-init-diff-warning

Menu entry

-

Command line syntax

`--no-for-init-diff-warning`

Description

Disable a warning that is issued when programs compiled without the **--old-for-init** option would have had different behavior under the old rules.

Related information

C++ compiler option **--old-for-init** (Use old for scoping rules)

C++ compiler option: --no-implicit-typename

Menu entry

-

Command line syntax

`--no-implicit-typename`

Default: implicit typename determination is enabled.

Description

Disable implicit determination, from context, whether a template parameter dependent name is a type or nontype.

Related information

C++ compiler option **--no-typename** (Disable the typename keyword)

C++ compiler option: --no-inlining

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Disable the option **Minimal inlining of function calls (C++)**.

Command line syntax

`--no-inlining`

Description

Disable minimal inlining of function calls.

Related information

-

C++ compiler option: --nonconst-ref-anachronism

Menu entry

-

Command line syntax

`--nonconst-ref-anachronism`

`--no-nonconst-ref-anachronism`

Default: **`--no-nonconst-ref-anachronism`**

Description

Enable or disable the anachronism of allowing a reference to nonconst to bind to a class rvalue of the right type. This anachronism is also enabled by the **--anachronisms** option.

Related information

C++ compiler option **--anachronisms** (Enable C++ anachronisms)

Section 2.2.3, *Anachronisms Accepted*

C++ compiler option: --nonstd-qualifier-deduction

Menu entry

-

Command line syntax

`--nonstd-qualifier-deduction`

Description

Controls whether non-standard template argument deduction should be performed in the qualifier portion of a qualified name. With this feature enabled, a template argument for the template parameter `T` can be deduced in contexts like `A<T>::B` or `T::B`. The standard deduction mechanism treats these as non-deduced contexts that use the values of template parameters that were either explicitly specified or deduced elsewhere.

Related information

-

C++ compiler option: --nonstd-using-decl

Menu entry

-

Command line syntax

`--nonstd-using-decl`

Default: non-standard using declarations are not allowed.

Description

Allow a non-member using declaration that specifies an unqualified name.

Related information

-

C++ compiler option: --no-parse-templates

Menu entry

-

Command line syntax

`--no-parse-templates`

Default: parsing of nonclass templates is enabled.

Description

Disable the parsing of nonclass templates in their generic form (i.e., even if they are not really instantiated). It is done by default if dependent name processing is enabled.

Related information

C++ compiler option **--no-dep-name** (Disable dependent name processing)

C++ compiler option: --no-pch-messages

Menu entry

-

Command line syntax

`--no-pch-messages`

Default: a message is displayed indicating that a precompiled header file was created or used in the current compilation. For example,

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

Description

Disable the display of a message indicating that a precompiled header file was created or used in the current compilation.

Related information

C++ compiler option **--pch** (Automatic PCH mode)

C++ compiler option **--use-pch** (Use precompiled header file)

C++ compiler option **--create-pch** (Create precompiled header file)

Section 2.10, *Precompiled Headers*

C++ compiler option: --no-preprocessing-only

Menu entry

Eclipse always does a full compilation.

Command line syntax

`--no-preprocessing-only`

Description

You can use this option in conjunction with the options that normally cause the C++ compiler to do preprocessing only (e.g., **--preprocess**, etc.) to specify that a full compilation should be done (not just preprocessing). When used with the implicit inclusion option, this makes it possible to generate a preprocessed output file that includes any implicitly included files.

Example

`cp166 --preprocess --implicit-include --no-preprocessing-only test.cc`

Related information

C++ compiler option **--preprocess** (Preprocessing only)

C++ compiler option **--implicit-include** (Implicit source file inclusion)

C++ compiler option: --no-stdinc / --no-stdstlinc

Menu entry

1.   Select **C/C++ Compiler » Miscellaneous**.

2.   Add the option **--no-stdinc** or **--no-stdstlinc** to the **Additional options** field.

Command line syntax

`--no-stdinc`

`--no-stdstlinc`

Description

With option **--no-stdinc** you tell the C++ compiler not to look in the default `include` directory relative to the installation directory, when searching for standard include files.

With option **--no-stdstlinc** you tell the C++ compiler not to look in the default `include.stl` directory relative to the installation directory, when searching for standard STL include files.

This way the C++ compiler only searches in the include file search paths you specified.

Related information

Section 5.2, *How the C++ Compiler Searches Include Files*

C++ compiler option: --no-tasking-sfr

Menu entry

1. Select **C/C++ Compiler » Preprocessing**.

2. Disable the option **Automatic inclusion of '.sfr' file**.

Command line syntax

```
--no-tasking-sfr
```

Description

Normally, the C++ compiler includes a special function register (SFR) file before compiling. The C++ compiler automatically selects the SFR file belonging to the target you selected on the **Processor** page (C compiler option **--cpu**).

With this option the C++ compiler does not include the register file reg*cpu*.sfr as based on the selected target processor.

Use this option if you want to use your own set of SFR files.

Related information

C++ compiler option **--cpu** (Select processor)

Section 1.3.5, *Accessing Hardware from C*

C++ compiler option: --no-typename

Menu entry

-

Command line syntax

`--no-typename`

Default: `typename` is recognized as a keyword.

Description

Disable recognition of the `typename` keyword.

Related information

C++ compiler option **--no-implicit-typename** (Disable implicit typename determination)

C++ compiler option: --no-use-before-set-warnings (-j)

Menu entry

1.   Select **C/C++ Compiler » Diagnostics**.

2.   Enable the option **Suppress C++ compiler "used before set" warnings**.

Command line syntax

`--no-use-before-set-warnings`

`-j`

Description

Suppress warnings on local automatic variables that are used before their values are set.

Related information

C++ compiler option **--no-warnings** (Suppress all warnings)

C++ compiler option: --no-warnings (-w)

Menu entry

1. Select **C/C++ Compiler » Diagnostics**.

2. Enable the option **Suppress all warnings**.

Command line syntax

`--no-warnings`

`-w`

Description

With this option you suppress all warning messages. Error messages are still issued.

Related information

C++ compiler option **--warnings-as-errors** (Treat warnings as errors)

C++ compiler option: --old-for-init

Menu entry

-

Command line syntax

`--old-for-init`

Description

Control the scope of a declaration in a `for-init-statement`. The old (cfront-compatible) scoping rules mean the declaration is in the scope to which the `for` statement itself belongs; the default (standard-conforming) rules in effect wrap the entire `for` statement in its own implicitly generated scope.

Related information

C++ compiler option **--no-for-init-diff-warning** (Disable warning for old for-scoping)

C++ compiler option: --old-line-commands

Menu entry

-

Command line syntax

`--old-line-commands`

Description

When generating source output, put out #line directives in the form # *nnn* instead of #line *nnn*.

Example

To do preprocessing only, without comments and with old style line control information, enter:

```
cp166 --preprocess --old-line-commands test.cc
```

Related information

C++ compiler option **--preprocess** (Preprocessing only)

C++ compiler option: --old-specializations

Menu entry

-

Command line syntax

`--old-specializations`

Description

Enable acceptance of old-style template specializations (that is, specializations that do not use the `template<>` syntax).

Related information

-

C++ compiler option: --option-file (-f)

Menu entry

-

Command line syntax

--option-file=*file*

-f *file*

Description

This option is primarily intended for command line use. Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the C++ compiler.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option **--option-file** multiple times.

Format of an option file

- Multiple arguments on one line in the option file are allowed.

- To include whitespace in an argument, surround the argument with single or double quotes.

- If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

  ```
  "This has a single quote ' embedded"

  'This has a double quote " embedded'

  'This has a double quote " and a single quote '"' embedded"
  ```

- When a text line reaches its length limit, use a **** to continue the line. Whitespace between quotes is preserved.

  ```
  "This is a continuation \
  line"

          -> "This is a continuation line"
  ```

- It is possible to nest command line files up to 25 levels.

Example

Suppose the file `myoptions` contains the following lines:

```
--embedded-c++
--define=DEMO=1
test.cc
```

Specify the option file to the C++ compiler:

```
cp166 --option-file=myoptions
```

This is equivalent to the following command line:

```
cp166 --embedded-c++ --define=DEMO=1 test.cc
```

Related information

-

C++ compiler option: --output (-o)

Menu entry

Eclipse names the output file always after the C++ source file.

Command line syntax

--output-file=_file_

-o _file_

Default: module name with `.ic` suffix.

Description

With this option you can specify another filename for the output file of the C++ compiler. Without this option the basename of the C++ source file is used with extension `.ic`.

You can also use this option in combination with the option **--preprocess (-E)** to redirect the preprocessing output to a file.

Example

To create the file `output.ic` instead of `test.ic`, enter:

```
cp166 --output=output.ic test.cc
```

To use the file `my.pre` as the preprocessing output file, enter:

```
cp166 --preprocess --output=my.pre test.cc
```

Related information

C++ compiler option **--preprocess** (Preprocessing)

C++ compiler option: --pch

Menu entry

1. Select **C/C++ Compiler » Precompiled C++ Headers**.

2. Enable the option **Automatically use/create precompiled header file**.

Command line syntax

`--pch`

Description

Automatically use and/or create a precompiled header file. If **--use-pch** or **--create-pch** (manual PCH mode) appears on the command line following this option, its effect is erased.

Related information

C++ compiler option **--use-pch** (Use precompiled header file)

C++ compiler option **--create-pch** (Create precompiled header file)

Section 2.10, *Precompiled Headers*

C++ compiler option: --pch-dir

Menu entry

1. Select **C/C++ Compiler » Precompiled C++ Headers**.

2. Enter a path in the **Precompiled header file directory**.

Command line syntax

--pch-dir=`directory-name`

Description

Specify the directory in which to search for and/or create a precompiled header file. This option may be used with automatic PCH mode (**--pch**) or manual PCH mode (**--create-pch** or **--use-pch**).

Example

To use the directory `/usr/include/pch` to automatically create precompiled header files, enter:

```
cp166 --pch-dir=/usr/include/pch --pch test.cc
```

Related information

C++ compiler option **--pch** (Automatic PCH mode)

C++ compiler option **--use-pch** (Use precompiled header file)

C++ compiler option **--create-pch** (Create precompiled header file)

Section 2.10, *Precompiled Headers*

C++ compiler option: --pch-verbose

Menu entry

-

Command line syntax

`--pch-verbose`

Description

In automatic PCH mode, for each precompiled header file that cannot be used for the current compilation, a message is displayed giving the reason that the file cannot be used.

Example

```
cp166 --pch --pch-verbose test.cc
```

Related information

C++ compiler option **--pch** (Automatic PCH mode)

Section 2.10, *Precompiled Headers*

C++ compiler option: --pending-instantiations

Menu entry

-

Command line syntax

`--pending-instantiations=`*n*

where *n* is the maximum number of instantiations of a single template.

Default: 64

Description

Specifies the maximum number of instantiations of a given template that may be in process of being instantiated at a given time. This is used to detect runaway recursive instantiations. If *n* is zero, there is no limit.

Example

To specify a maximum of 32 pending instantiations, enter:

```
cp166 --pending-instantiations=32 test.cc
```

Related information

Section 2.5, *Template Instantiation*

C++ compiler option: --preprocess (-E)

Menu entry

1.   Select **C/C++ Compiler » Preprocessing**.

2.   Enable the option **Store preprocessor output in <file>.pre**.

3.   (Optional) Enable the option **Keep comments in preprocessor output**.

4.   (Optional) Enable the option **Keep #line info in preprocessor output**.

Command line syntax

--preprocess[**=**_flags_]

-E[_flags_]

You can set the following flags:

+/-comments	**c/C**	keep comments
+/-make	**m/M**	generate dependencies for make
+/-noline	**p/P**	strip #line source position information

Default: **-ECMP**

Description

With this option you tell the C++ compiler to preprocess the C++ source. Under Eclipse the C++ compiler sends the preprocessed output to the file _name_.pre (where _name_ is the name of the C++ source file to compile). Eclipse also compiles the C++ source.

On the command line, the C++ compiler sends the preprocessed file to stdout. To capture the information in a file, specify an output file with the option **--output**.

With **--preprocess=+comments** you tell the preprocessor to keep the comments from the C++ source file in the preprocessed output.

With **--preprocess=+make** the C++ compiler will generate dependency lines that can be used in a Makefile. The preprocessor output is discarded. The default target name is the basename of the input file, with the extension .obj. With the option **--make-target** you can specify a target name which overrules the default target name.

> When implicit inclusion of templates is enabled, the output may indicate false (but safe) dependencies unless **--no-preprocessing-only** is also used.

With **--preprocess=+noline** you tell the preprocessor to strip the #line source position information (lines starting with #line). These lines are normally processed by the assembler and not needed in the preprocessed output. When you leave these lines out, the output is easier to read.

Example

```
cp166 --preprocess=+comments,-make,-noline test.cc --output=test.pre
```

The C++ compiler preprocesses the file `test.cc` and sends the output to the file `test.pre`. Comments are included but no dependencies are generated and the line source position information is not stripped from the output file.

Related information

C++ compiler option **--no-preprocessing-only** (Force full compilation)

C++ compiler option **--dep-file** (Generate dependencies in a file)

C++ compiler option **--make-target** (Specify target name for **-Em** output)

C++ compiler option: --remarks (-r)

Menu entry

1. Select **C/C++ Compiler » Diagnostics**.

2. Enable the option **Issue remarks on C++ code**.

Command line syntax

```
--remarks
```

```
-r
```

Description

Issue remarks, which are diagnostic messages even milder than warnings.

Related information

Section 5.3, *C++ Compiler Error Messages*

C++ compiler option: --remove-unneeded-entities

Menu entry

-

Command line syntax

```
--remove-unneeded-entities
```

Description

Enable an optimization to remove types, variables, routines, and related constructs that are not really needed. Something may be referenced but unneeded if it is referenced only by something that is itself unneeded; certain entities, such as global variables and routines defined in the translation unit, are always considered to be needed.

Related information

-

C++ compiler option: --rtti

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Support for C++ RTTI (run-time type information)**.

Command line syntax

`--rtti`

Default: RTTI (run-time type information) features are disabled.

Description

Enable support for RTTI (run-time type information) features: `dynamic_cast`, `typeid`.

The macro `__RTTI` is defined when RTTI support is enabled.

Related information

-

C++ compiler option: --schar (-s)

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Disable the option **Treat "char" variables as unsigned**.

Command line syntax

`--schar`

`-s`

Description

With this option `char` is the same as `signed char`.

When plain `char` is signed, the macro `__SIGNED_CHARS__` is defined.

Related information

C++ compiler option **--uchar** (Plain char is unsigned)

Section 1.1, *Data Types*

C++ compiler option: --signed-bitfields

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Treat "int" bit-fields as signed**.

Command line syntax

```
--signed-bitfields
```

Description

For bit-fields it depends on the implementation whether a plain `int` is treated as `signed int` or `unsigned int`. By default an `int` bit-field is treated as `unsigned int`. This offers the best performance. With this option you tell the C++ compiler to treat `int` bit-fields as `signed int`. In this case, you can still add the keyword `unsigned` to treat a particular `int` bit-field as `unsigned`.

Related information

C compiler option **--signed-bitfields**

Section 1.1, *Data Types*

C++ compiler option: --special-subscript-cost

Menu entry

-

Command line syntax

`--special-subscript-cost`

Description

Enable a special nonstandard weighting of the conversion to the integral operand of the [ ] operator in overload resolution.

This is a compatibility feature that may be useful with some existing code. With this feature enabled, the following code compiles without error:

```
struct A {
    A();
    operator int *();
    int operator[](unsigned);
};
void main() {
    A a;
    a[0];   // Ambiguous, but allowed with this option
            // operator[] is chosen
}
```

Related information

-

C++ compiler option: --strict (-A)

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Disable the option **Allow non-ANSI/ISO C++ features**.

Command line syntax

```
--strict
```

```
-A
```

Default: non-ANSI/ISO C++ features are enabled.

Description

Enable strict ANSI/ISO mode, which provides diagnostic messages when non-standard features are used, and disables features that conflict with ANSI/ISO C or C++. All ANSI/ISO violations are issued as errors.

Example

To enable strict ANSI mode, with error diagnostic messages, enter:

```
cp166 --strict test.cc
```

Related information

C++ compiler option **--strict-warnings** (Strict ANSI/ISO mode with warnings)

C++ compiler option: --strict-warnings (-a)

Menu entry

-

Command line syntax

`--strict-warnings`

`-a`

Default: non-ANSI/ISO C++ features are enabled.

Description

This option is similar to the option **--strict**, but all violations are issued as warnings instead of errors.

Example

To enable strict ANSI mode, with warning diagnostic messages, enter:

```
cp166 --strict-warnings test.cc
```

Related information

C++ compiler option **--strict** (Strict ANSI/ISO mode with errors)

C++ compiler option: --suppress-vtbl

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Enable the option **Suppress definition of virtual function tables (C++)**.

Command line syntax

```
--suppress-vtbl
```

Description

Suppress definition of virtual function tables in cases where the heuristic used by the C++ compiler to decide on definition of virtual function tables provides no guidance. The virtual function table for a class is defined in a compilation if the compilation contains a definition of the first non-inline non-pure virtual function of the class. For classes that contain no such function, the default behavior is to define the virtual function table (but to define it as a local static entity). The **--suppress-vtbl** option suppresses the definition of the virtual function tables for such classes, and the **--force-vtbl** option forces the definition of the virtual function table for such classes. **--force-vtbl** differs from the default behavior in that it does not force the definition to be local.

Related information

C++ compiler option **--force-vtbl** (Force definition of virtual function tables)

C++ compiler option: --sys-include

Menu entry

-

Command line syntax

--sys-include=*directory*,...

Description

Change the algorithm for searching system include files whose names do not have an absolute pathname to look in *directory*.

Example

To add the directory /proj/include to the system include file search path, enter:

```
cp166 --sys-include=/proj/include test.cc
```

Related information

C++ compiler option **--include-directory** (Add directory to include file search path)

Section 5.2, *How the C++ Compiler Searches Include Files*

C++ compiler option: --template-directory

Menu entry

-

Command line syntax

--template-directory=$directory$,...

Description

Specifies a directory name to be placed on the exported template search path. The directories are used to find the definitions of exported templates (.et files) and are searched in the order in which they are specified on the command line. The current directory is always the first entry on the search path.

Example

To add the directory export to the exported template search path, enter:

```
cp166 --template-directory=export test.cc
```

Related information

Section 2.5.5, *Exported Templates*

C++ compiler option: --timing

Menu entry

-

Command line syntax

`--timing`

Default: no timing information is generated.

Description

Generate compilation timing information. This option causes the C++ compiler to display the amount of CPU time and elapsed time used by each phase of the compilation and a total for the entire compilation.

Example

```
cp166 --timing test.cc
```

```
processed 180 lines at 8102 lines/min
```

Related information

-

C++ compiler option: --trace-includes

Menu entry

-

Command line syntax

`--trace-includes`

Description

Output a list of the names of files #included to the error output file. The source file is compiled normally (i.e. it is not just preprocessed) unless another option that causes preprocessing only is specified.

Example

```
cp166 --trace-includes test.cc

iostream.h
string.h
```

Related information

C++ compiler option **--preprocess** (Preprocessing only)

C++ compiler option: --type-traits-helpers

Menu entry

-

Command line syntax

--type-traits-helpers

--no-type-traits-helpers

Default: in C++ mode type traits helpers are enabled by default. In GNU C++ mode, type traits helpers are never enabled by default.

Description

Enable or disable type traits helpers (like `__is_union` and `__has_virtual_destructor`). Type traits helpers are meant to ease the implementation of ISO/IEC TR 19768.

The macro `__TYPE_TRAITS_ENABLED` is defined when type traits pseudo-functions are enabled.

Related information

-

C++ compiler option: --uchar (-u)

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Treat "char" variables as unsigned**.

Command line syntax

```
--uchar
```

```
-u
```

Description

By default char is the same as specifying signed char. With this option char is the same as unsigned char.

Related information

C++ compiler option **--schar** (Plain char is signed)

Section 1.1, *Data Types*

C++ compiler option: --undefine (-U)

Menu entry

1. Select **C/C++ Compiler » Preprocessing**

    The Defined symbols box shows the symbols that are currently defined.

2. To remove a defined symbol, select the symbol in the **Defined symbols** box and click on the **Delete** button.

Command line syntax

`--undefine=`*macro_name*

`-U`*macro_name*

Description

Remove any initial definition of *macro_name* as in `#undef`. **--undefine** options are processed after all **--define** options have been processed.

You cannot undefine a predefined macro as specified in Section 2.9, *Predefined Macros*, except for:

    __STDC__

    __cplusplus

    __SIGNED_CHARS__

Example

To undefine the predefined macro `__cplusplus`:

`cp166 --undefine=__cplusplus test.cc`

Related information

C++ compiler option **--define** (Define preprocessor macro)

Section 2.9, *Predefined Macros*

C++ compiler option: --use-pch

Menu entry

1.   Select **C/C++ Compiler » Precompiled C++ Headers**.

2.   Enter a filename in the **Use precompiled header file** field.

Command line syntax

--use-pch=_filename_

Description

Use a precompiled header file of the specified name as part of the current compilation. If **--pch** (automatic PCH mode) or **--create-pch** appears on the command line following this option, its effect is erased.

Example

To use the precompiled header file with the name test.pch, enter:

```
cp166 --use-pch=test.pch test.cc
```

Related information

C++ compiler option **--pch** (Automatic PCH mode)

C++ compiler option **--create-pch** (Create precompiled header file)

Section 2.10, *Precompiled Headers*

C++ compiler option: --user-stack

Menu entry

1. Select **C/C++ Compiler » Memory Model**.

2. Enable the option **Use user stack for return addresses**.

Command line syntax

`--user-stack`

Description

With this option the function return address is stored on the user stack instead of on the system stack. Also the user stack run-time libraries are used and the user stack calling convention is used to call the run-time library functions.

Related information

C compiler option **--user-stack**

C++ compiler option: --using-std

Menu entry

-

Command line syntax

`--using-std`

Default: implicit use of the `std` namespace is disabled.

Description

Enable implicit use of the `std` namespace when standard header files are included. Note that this does not do the equivalent of putting a `"using namespace std;"` in the program to allow old programs to be compiled with new header files; it has a special and localized meaning related to the TASKING versions of certain header files, and is unlikely to be of much use to end-users of the TASKING C++ compiler.

Related information

C++ compiler option **--namespaces** (Support for namespaces)

Section 2.4, *Namespace Support*

C++ compiler option: --variadic-macros

Menu entry

-

Command line syntax

`--variadic-macros`

Default: macros with a variable number of arguments are not allowed.

Description

Allow macros with a variable number of arguments.

Related information

C++ compiler option **--extended-variadic-macros** (Allow extended variadic macros)

C++ compiler option: --version (-V)

Menu entry

-

Command line syntax

`--version`

`-V`

Description

Display version information. The C++ compiler ignores all other options or input files.

Example

```
cp166 --version
```

The C++ compiler does not compile any files but displays the following version information:

```
TASKING VX-toolset for C166: C++ compiler    vx.yrz Build nnn
Copyright 2004-year Altium BV              Serial# 00000000
```

Related information

-

C++ compiler option: --warnings-as-errors

Menu entry

1. Select **Global Options**.

2. Enable the option **Treat warnings as errors**.

Command line syntax

`--warnings-as-errors`[`=`*number*`,...`]

Description

If the C++ compiler encounters an error, it stops compiling. When you use this option without arguments, you tell the C++ compiler to treat all warnings as errors. This means that the exit status of the C++ compiler will be non-zero after one or more compiler warnings. As a consequence, the C++ compiler now also stops after encountering a warning.

You can also limit this option to specific warnings by specifying a comma-separated list of warning numbers.

Related information

C++ compiler option **--no-warnings** (Suppress all warnings)

C++ compiler option: --wchar_t-keyword

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Allow the 'wchar_t' keyword (C++)**.

Command line syntax

`--wchar_t-keyword`

Default: `wchar_t` is not recognized as a keyword.

Description

Enable recognition of `wchar_t` as a keyword.

The macro `_WCHAR_T` is defined when `wchar_t` is recognized as a keyword.

Related information

-

C++ compiler option: --xref-file (-X)

Menu entry

-

Command line syntax

--xref-file=*file*

-X*file*

Description

Generate cross-reference information in a *file*. For each reference to an identifier in the source program, a line of the form

```
symbol_id name X file-name line-number column-number
```

is written, where *X* is

D	for definition;
d	for declaration (that is, a declaration that is not a definition);
M	for modification;
A	for address taken;
U	for used;
C	for changed (but actually meaning used and modified in a single operation, such as an increment);
R	for any other kind of reference, or
E	for an error in which the kind of reference is indeterminate.

symbol-id is a unique decimal number for the symbol. The fields of the above line are separated by tab characters.

Related information

-

12.4. Assembler Options

This section lists all assembler options.

Options in Eclipse versus options on the command line

Most command line options have an equivalent option in Eclipse but some options are only available on the command line. Eclipse invokes the assembler via the control program. Therefore, it uses the syntax of the control program to pass options and files to the assembler. If there is no equivalent option in Eclipse, you can specify a command line option in Eclipse as follows:

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

   The Properties dialog appears.

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

   In the right pane the Settings appear.

3. On the Tool Settings tab, select **Assembler » Miscellaneous**.

4. In the **Additional options** field, enter one or more command line options.

   *Because Eclipse uses the control program, Eclipse automatically precedes the option with **-Wa** to pass the option via the control program directly to the assembler.*

> Note that the options you enter in the Assembler page are not only used for hand-coded assembly files, but also for the assembly files generated by the compiler.

Be aware that some command line options are not useful in Eclipse or just do not have any effect. For example, the option **-V** displays version header information and has no effect in Eclipse.

Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (**-**) character, long option names always begin with two minus (**--**) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

Options can have flags or suboptions. To switch a flag 'on', use a lowercase letter or a **+***longflag*. To switch a flag off, use an uppercase letter or a **-***longflag*. Separate *longflags* with commas. The following two invocations are equivalent:

```
as166 -Ogs test.src
as166 --optimize=+generics,+instr-size test.src
```

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

Assembler option: --case-insensitive (-c)

Menu entry

1. Select **Assembler » Symbols**.

2. Enable the option **Case insensitive identifiers**.

Command line syntax

`--case-insensitive`

`-c`

Default: case sensitive

Description

With this option you tell the assembler not to distinguish between upper and lower case characters. By default the assembler considers upper and lower case characters as different characters.

Assembly source files that are generated by the compiler must always be assembled case sensitive. When you are writing your own assembly code, you may want to specify the case insensitive mode.

Example

When assembling case insensitive, the label `LabelName` is the same label as `labelname`.

`as166 --case-insensitive test.src`

Related information

-

Assembler option: --check

Menu entry

-

Command line syntax

`--check`

Description

With this option you can check the source code for syntax errors, without generating code. This saves time in developing your application.

The assembler reports any warnings and/or errors.

This option is available on the command line only.

Related information

C compiler option **--check** (Check syntax)

Assembler option: --compatibility

Menu entry

-

Command line syntax

--compatibility=*flag*,...

You can specify the following argument:

sectionlabels   Generate section name as label

Description

With this option you can adjust aspects of the assembler for backwards compatibility with the classic TASKING C166/ST10 toolset. With the argument **sectionlabels** a section name can also be used as a label instead of just as a symbol.

Example

as166 --compatibility=sectionlabels test.src

Related information

Assembler directive **.SECTION**

Assembler option: --core

Menu entry

1. Expand **C/C++ Build** and select **Processor**.

2. From the **Processor selection** list, expand **Custom** and select a core.

Command line syntax

--core=<code>*core*</code>

You can specify the following *core* arguments:

c16x	C16x instruction set
st10	ST10 instruction set
st10mac	ST10 with MAC co-processor support
c166sv1	C166S v1.0 support
xc16x	XC16x/XE16xx/XC2xxx instruction set
super10	Super10 instruction set
super10m345	Enhanced Super10M345 instruction set

Default: derived from **--cpu** if specified and known or else **--core=xc16x**

Description

With this option you specify the core architecture (instruction set) for a target processor for which you create your application. If **--cpu** is specified and the supplied CPU is known by the assembler, the assembler selects the correct core automatically.

For more information see assembler option **--cpu**.

Example

Specify a custom core:

```
as166 --core=st10 test.src
```

Related information

Assembler option **--cpu** (Select processor)

Assembler option: --cpu (-C)

Menu entry

1. Expand **C/C++ Build** and select **Processor**.

2. From the **Processor selection** list, select a processor or expand **Custom** and select a core.

Command line syntax

--cpu=*cpu*

-C*cpu*

Description

With this option you define the target processor for which you create your application.

Based on this option the assembler always includes the special function register file reg*cpu*.sfr, unless you specify option **--no-tasking-sfr**.

If you use the control program to compile C or C++ files, the control program will not pass the **--cpu** option to the assembler, but only the **--core** option. The reason for this is that in general all SFRs are already processed by the C compiler, and there are no SFRs left in the assembly code. This speeds up the build time, because the assembler does not need to read the SFR file. When you use in-line assembly in your C code that uses SFRs, you do want the assembler to read the SFR file. In this case you need to pass the **--cpu** option directly to the assembler (control program option **--pass-assembler=--cpu=***cpu* / **-Wa-C***cpu*). It is recommended to use this option only for those modules that use in-line assembly. This avoids the impact to the build speed to a minimum.

Note that for .asm and .src files the control program will pass the **--cpu** option to the assembler.

If you select a target from the list, the core is known. If you specify a Custom processor, you need to select the core that matches the core of your custom processor (option **--core**). The assembler knows all CPUs with core c16x, st10, st10mac and super10. If you specify a CPU that is not in that list, it is assumed to be an xc16x.

The following table show the relation between the two options:

--cpu=*cpu*	**--core=***core*	*core*	**Register file**
no	no	c16x	c166, cp166: none as166: regc16x.sfr
no	yes	*core*	c166, cp166: none as166: reg*core*.sfr
yes	no	derived from *cpu* for core c16x, st10, st10mac and super10 If the *cpu* is unknown core xc16x is assumed	reg*cpu*.sfr
yes	yes	*core*	reg*cpu*.sfr

The standard list of supported processors is defined in the file `processors.xml`. This file defines for each processor its full name (for example, XC2287-72F), the base CPU name (for example, xc2287), the core settings (for example, xc16x), the on-chip flash settings, the list of silicon bugs for that processor. Each processor also defines an option to supply to the linker for preprocessing the LSL file for the applicable on-chip memory definitions. The option is for example **-DXC2287_72M**.

Example

Specify an existing processor:

```
as166 --cpu=c167cs40 test.src
```

Specify a custom processor:

```
as166 --cpu=custom --core=st10 test.src
```

Related information

Assembler option **--core** (Select the core)

Assembler option **--no-tasking-sfr** (Do not include SFR file)

Section 3.6, *Special Function Registers*

Assembler option: --debug-info (-g)

Menu entry

1. Select **Assembler » Symbols**.

2. Select an option from the **Generate symbolic debug** list.

Command line syntax

`--debug-info`[**`=`***flags*]

`-g`[*flags*]

You can set the following flags:

+/-asm	**a/A**	Assembly source line information
+/-hll	**h/H**	Pass high level language debug information (HLL)
+/-local	**l/L**	Assembler local symbols debug information
+/-smart	**s/S**	Smart debug information

Default: **`--debug-info=+hll`**

Default (without flags): **`--debug-info=+smart`**

Description

With this option you tell the assembler which kind of debug information to emit in the object file.

You cannot specify **--debug-info=+asm,+hll**. Either the assembler generates assembly source line information, or it passes HLL debug information.

When you specify **--debug-info=+smart**, the assembler selects which flags to use. If high level language information is available in the source file, the assembler passes this information (same as **--debug-info=-asm,+hll,-local**). If not, the assembler generates assembly source line information (same as **--debug-info=+asm,-hll,+local**).

With **--debug-info=AHLS** the assembler does not generate any debug information.

Related information

Assembler control **$DEBUG**

Assembler option: --define (-D)

Menu entry

1. Select **Assembler » Preprocessing**.

    The Defined symbols box right-below shows the symbols that are currently defined.

2. To define a new symbol, click on the **Add** button in the **Defined symbols** box.

3. Type the symbol definition (for example, `demo=1`)

> Use the **Edit** and **Delete** button to change a macro definition or to remove a macro from the list.

Command line syntax

--define=`macro_name`[**=**`macro_definition`]

-D`macro_name`[**=**`macro_definition`]

Description

With this option you can define a macro and specify it to the assembler preprocessor. If you only specify a macro name (no macro definition), the macro expands as '1'.

You can specify as many macros as you like. Simply use the **Add** button to add new macro definitions.

On the command line, use the option **--define** (**-D**) multiple times. If the command line exceeds the limit of the operating system, you can define the macros in an option file which you then must specify to the assembler with the option **--option-file** (**-f**) *file*.

Defining macros with this option (instead of in the assembly source) is, for example, useful in combination with conditional assembly as shown in the example below.

> This option has the same effect as defining symbols via the `.DEFINE`, `.SET`, and `.EQU` directives. (similar to `#define` in the C language). With the `.MACRO` directive you can define more complex macros.

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:

```
as166 --define=DEMO test.src
as166 --define=DEMO=1 test.src
```

Note that both invocations have the same effect.

Related information

Assembler option **--option-file** (Specify an option file)

Assembler option: --dep-file

Menu entry

-

Command line syntax

--dep-file[**=***file*]

Description

With this option you tell the assembler to generate dependency lines that can be used in a Makefile. The dependency information will be generated in addition to the normal output file.

By default, the information is written to a file with extension `.d`. When you specify a filename, all dependencies will be combined in the specified file.

Example

```
as166 --dep-file=test.dep test.src
```

The assembler assembles the file `test.src`, which results in the output file `test.obj`, and generates dependency lines in the file `test.dep`.

Related information

Assembler option **--make-target** (Specify target name for **--dep-file** output)

Assembler option: --diag

Menu entry

1. From the **Window** menu, select **Show View » Other » Basic » Problems**.

    The Problems view is added to the current perspective.

2. In the Problems view right-click on a message.

    A popup menu appears.

3. Select **Detailed Diagnostics Info**.

    A dialog box appears with additional information.

Command line syntax

--diag=[*format*:]{**all** | *nr*,...}

You can set the following output formats:

html	HTML output.
rtf	Rich Text Format.
text	ASCII text.

Default format: text

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to stdout (normally your screen) and in the format you specify. You can specify the following formats: **html**, **rtf** or **text** (default). To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas.

Example

To display an explanation of message number 244, enter:

```
as166 --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:

```
as166 --diag=html:all > aserrors.html
```

Related information

Section 7.6, *Assembler Error Messages*

Assembler option: --emit-locals

Menu entry

1. Select **Assembler » Symbols**.

2. Enable or disable one or both of the following options:

   • Emit local EQU symbols

   • Emit local non-EQU symbols

Command line syntax

`--emit-locals[=`*flag*`,...]`

You can set the following flags:

+/-equs	**e/E**	emit local EQU symbols
+/-symbols	**s/S**	emit local non-EQU symbols

Default: `--emit-locals=ES`

Default (without flags): `--emit-locals=+symbols`

Description

With the option **--emit-locals=+equs** the assembler also emits local EQU symbols to the object file. Normally, only global symbols and non-EQU local symbols are emitted. Having local symbols in the object file can be useful for debugging.

Related information

Assembler directive **.EQU**

Assembler option: --error-file

Menu entry

-

Command line syntax

--error-file[**=***file*]

Description

With this option the assembler redirects 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:

```
as166 --error-file=errors.ers test.src
```

Related information

Section 7.6, *Assembler Error Messages*

Assembler option: --error-limit

Menu entry

1. Select **Assembler » Diagnostics**.

2. Enter a value in the **Maximum number of emitted errors** field.

Command line syntax

`--error-limit=`*number*

Default: 42

Description

With this option you tell the assembler to only emit the specified maximum number of errors. When 0 (null) is specified, the assembler emits all errors. Without this option the maximum number of errors is 42.

Related information

Section 7.6, *Assembler Error Messages*

Assembler option: --help (-?)

Menu entry

-

Command line syntax

--help[**=**_item_]

-?

You can specify the following arguments:

options   Show extended option descriptions

Description

Displays an overview of all command line options. When you specify the argument **options** you can list detailed option descriptions.

Example

The following invocations all display a list of the available command line options:

```
as166 -?
as166 --help
as166
```

To see a detailed description of the available options, enter:

```
as166 --help=options
```

Related information

-

Assembler option: --include-directory (-I)

Menu entry

1. Select **Assembler » Include Paths**.

    The Include paths box shows the directories that are added to the search path for include files.

2. To define a new directory for the search path, click on the **Add** button in the **Include paths** box.

3. Type or select a path.

Use the **Edit** and **Delete** button to change a path or to remove a path from the list.

Command line syntax

--include-directory=*path*,...

-I*path*,...

Description

With this option you can specify the path where your include files are located. A relative path will be relative to the current directory,

The order in which the assembler searches for include files is:

1. The pathname in the assembly file and the directory of the assembly source.

2. The path that is specified with this option.

3. The path that is specified in the environment variable `AS166INC` 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:

```
as166 --include-directory=c:\proj\include test.src
```

First the assembler looks for the file `myinc.inc` in the directory where `test.src` is located. If it does not find the file, it looks in the directory `c:\proj\include` (this option). If the file is still not found, the assembler searches in the environment variable and then in the default include directory.

Related information

Assembler option **--include-file** (Include file at the start of the input file)

Assembler option: --include-file (-H)

Menu entry

1. Select **Assembler » Preprocessing**.

   The upper box shows the files that are currently included before the assembling starts.

2. To define a new file, click on the **Add** button in the **Include files at start of input file** box.

3. Type the full path and file name or select a file.

Use the **Edit** and **Delete** button to change a file name or to remove a file from the list.

Command line syntax

```
--include-file=file,...
```

```
-Hfile,...
```

Description

With this option (set at project level) you include one extra file at the beginning of the assembly source file. The specified include file is included before all other includes. This is the same as specifying `.INCLUDE 'file'` at the beginning of your assembly source.

Example

```
as166 --include-file=myinc.inc test.src
```

The file `myinc.inc` is included at the beginning of `test.src` before it is assembled.

Related information

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

Assembler option: --keep-output-files (-k)

Menu entry

Eclipse *always* removes the object file when errors occur during assembling.

Command line syntax

```
--keep-output-files
```

```
-k
```

Description

If an error occurs during assembling, the resulting object file (`.obj`) may be incomplete or incorrect. With this option you keep the generated object file when an error occurs.

By default the assembler removes the generated object file when an error occurs. This is useful when you use the make utility. If the erroneous files are not removed, the make utility may process corrupt files on a subsequent invocation.

Use this option when you still want to use the generated object. For example when you know that a particular error does not result in a corrupt object file.

Related information

Assembler option **--warnings-as-errors** (Treat warnings as errors)

Assembler option: --list-file (-l)

Menu entry

1. Select **Assembler » List File**.

2. Enable the option **Generate list file**.

3. Enable or disable the types of information to be included.

Command line syntax

--list-file[**=***file*]

-l[*file*]

Default: no list file is generated

Description

With this option you tell the assembler to generate a list file. A list file shows the generated object code and the relative addresses. Note that the assembler generates a relocatable object file with relative addresses.

With the optional *file* you can specify an alternative name for the list file. By default, the name of the list file is the basename of the source file with the extension .lst.

Related information

Assembler option **--list-format** (Format list file)

Assembler option: --list-format (-L)

Menu entry

1. Select **Assembler » List File**.

2. Enable the option **Generate list file**.

3. Enable or disable the types of information to be included.

Command line syntax

--list-format=$flag$**,...**

-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 #line directives
+/-macro	**m/M**	List macro definition
+/-empty-line	**n/N**	List empty source lines and comment lines
+/-conditional	**p/P**	List conditional assembly
+/-equate	**q/Q**	List equate and set directives (.EQU, .SET)
+/-relocations	**r/R**	List relocations characters 'r'
+/-hll	**s/S**	List HLL symbolic debug informations
+/-equate-values	**v/V**	List equate and set values
+/-wrap-lines	**w/W**	Wrap source lines
+/-macro-expansion	**x/X**	List macro expansions
+/-cycle-count	**y/Y**	List cycle counts
+/-define-expansion	**z/Z**	List define expansions

Use the following options for predefined sets of flags:

--list-format=0	**-L0**	All options disabled Alias for **--list-format=DEGILMNPQRSVWXYZ**
--list-format=1	**-L1**	All options enabled Alias for **--list-format=degilmnpqrsvwxyz**

Default: **--list-format=dEGilMnPqrsVwXyZ**

Description

With this option you specify which information you want to include in the list file.

On the command line you must use this option in combination with the option **--list-file** (**-l**).

Related information

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

Assembler option **--section-info=+list** (Display section information in list file)

Assembler option: --make-target

Menu entry

-

Command line syntax

`--make-target=`*name*

Description

With this option you can overrule the default target name in the make dependencies generated by the option **--dep-file**. The default target name is the basename of the input file, with extension `.obj`.

Example

```
as166 --dep-file --make-target=../mytarget.obj test.src
```

The assembler generates dependency lines with the default target name `../mytarget.obj` instead of `test.obj`.

Related information

Assembler option **--dep-file** (Generate dependencies in a file)

Assembler option: --nested-sections (-N)

Menu entry

1. Select **Assembler » Miscellaneous**.

2. Enable the option **Allow nested sections**.

Command line syntax

```
--nested-sections
```

```
-N
```

Description

With this option it is allowed to have nested sections in your assembly source file. When you use this option every .SECTION directive must have a corresponding .ENDS directive.

Example

```
CSEC1  .SECTION far
   ; section
CSEC2  .SECTION far
   ; nested section
CSEC2  .ENDS
CSEC1  .ENDS
```

Related information

Assembler directive **.SECTION**

Assembler option: --no-bit-rewrites

Menu entry

1.   Select **Assembler » Optimization**.

2.   Disable the option **Allow bit-jump rewrites**.

Command line syntax

```
--no-bit-rewrites
```

Description

With this option you instruct the assembler that rewrites of JB and JNB instructions to out-of-range labels are not allowed. This will cause errors on out-of-range JB and JNB instructions.

Related information

-

Assembler option: --no-tasking-sfr

Menu entry

1. Select **Assembler » Preprocessing**.

2. Disable the option **Automatic inclusion of '.sfr' file**.

Command line syntax

`--no-tasking-sfr`

Description

Normally, the assembler includes a special function register (SFR) file before assembling. The assembler automatically selects the SFR file belonging to the target you select on the **Processor** page (assembler option **--cpu**).

With this option the assembler does not include the register file reg*cpu*.sfr as based on the selected target processor.

Use this option if you want to use your own set of SFR files.

Related information

Assembler option **--cpu** (Select processor)

Assembler option: --no-warnings (-w)

Menu entry

1. Select **Assembler » Diagnostics**.

    The Suppress warnings box shows the warnings that are currently suppressed.

2. To suppress a warning, click on the **Add** button in the **Suppress warnings** box.

3. Enter the numbers, separated by commas, of the warnings you want to suppress (for example `201,202`). Or you can use the **Add** button multiple times.

4. To suppress all warnings, enable the option **Suppress all warnings**.

Use the **Edit** and **Delete** button to change a warning number or to remove a number from the list.

Command line syntax

--no-warnings[**=**_number_,...]

-w[_number_,...]

Description

With this option you can suppresses all warning messages or specific warning messages.

On the command line this option works as follows:

*   If you do not specify this option, all warnings are reported.

*   If you specify this option but without numbers, all warnings are suppressed.

*   If you specify this option with a number, only the specified warning is suppressed. You can specify the option **--no-warnings=**_number_ multiple times.

Example

To suppress warnings 201 and 202, enter:

```
as166 test.src --no-warnings=201,202
```

Related information

Assembler option **--warnings-as-errors** (Treat warnings as errors)

Assembler option: --optimize (-O)

Menu entry

1. Select **Assembler » Optimization**.

2. Select one or more of the following options:

   - Allow JMPA+/JMPA- for speed optimization

   - Optimize generic instructions

   - Optimize jump chains

   - Optimize instruction size

Command line syntax

--optimize=`flag`**,...**

-O`flags`

You can set the following flags:

+/-jumpa-speed	**a/A**	Allow JMPA+/JMPA- for speed optimization
+/-generics	**g/G**	Allow generic instructions
+/-jumpchains	**j/J**	Optimize jump chains
+/-instr-size	**s/S**	Optimize instruction size

Default: **--optimize=agJs**

Description

With this option you can control the level of optimization. For details about each optimization see
Section 7.4, *Assembler Optimizations*

When you use this option to specify a set of optimizations, you can turn on or off the optimizations in your
assembly source file with the assembler controls $optimize/$nooptimize.

Related information

Assembler control **$OPTIMIZE**

Section 7.4, *Assembler Optimizations*

Assembler option: --option-file (-f)

Menu entry

1. Select **Assembler » Miscellaneous**.

2. Add the option **--option-file** to the **Additional options** field.

   Be aware that the options in the option file are added to the assembler options you have set in the other pages. Only in extraordinary cases you may want to use them in combination.

Command line syntax

--option-file=*file***,...**

-f *file***,...**

Description

This option is primarily intended for command line use. Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the assembler.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

Option files can also be generated on the fly, for example by the make utility. You can specify the option **--option-file** multiple times.

Format of an option file

- Multiple arguments on one line in the option file are allowed.

- To include whitespace in an argument, surround the argument with single or double quotes.

- If you want to use single quotes as part of the argument, surround the argument by double quotes and 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:

```
as166 --option-file=myoptions
```

This is equivalent to the following command line:

```
as166 --debug=+asm,-local test.src
```

Related information

-

Assembler option: --output (-o)

Menu entry

Eclipse names the output file always after the input file.

Command line syntax

--output=_file_

-o _file_

Description

With this option you can specify another filename for the output file of the assembler. Without this option, the basename of the assembly source file is used with extension .obj.

Example

To create the file relobj.obj instead of asm.obj, enter:

as166 --output=relobj.obj asm.src

Related information

-

Assembler option: --page-length

Menu entry

1. Select **Assembler » Miscellaneous**.

2. Add the option **--page-length** to the **Additional options** field.

Command line syntax

--page-length=_number_

Default: 72

Description

If you generate a list file with the assembler option **--list-file**, this option sets the number of lines in a page in the list file. The default is 72, the minimum is 10. As a special case, a page length of 0 turns off page breaks.

Related information

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

Assembler control **$PAGELENGTH**

Assembler option: --page-width

Menu entry

1. Select **Assembler » Miscellaneous**.

2. Add the option **--page-width** to the **Additional options** field.

Command line syntax

`--page-width=`*number*

Default: 132

Description

If you generate a list file with the assembler option **--list-file**, this option sets the number of columns per line on a page in the list file. The default is 132, the minimum is 40.

Related information

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

Assembler control **$PAGEWIDTH**

Assembler option: --preprocess (-E)

Menu entry

-

Command line syntax

`--preprocess`

`-E`

Description

With this option the assembler will only preprocess the assembly source file. The assembler sends the preprocessed file to stdout.

Related information

-

Assembler option: --preprocessor-type (-m)

Menu entry

-

Command line syntax

`--preprocessor-type=`*type*

`-m`*type*

You can set the following preprocessor types:

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: --require-end

Menu entry

-

Command line syntax

`--require-end`

Description

With this option the assembly source must be terminated with the `.END` directive.

Related information

Assembler directive **.END**

Assembler option: --retcheck

Menu entry

1. Select **Assembler » Diagnostics**.

2. Enable the option **Check if return instructions match procedure type**.

Command line syntax

```
--retcheck
```

Description

With this option the assembler can check if a return statement is present in procedures, whether the program flow stops at the end of a procedure and whether the type of return instruction used is correct for the type of procedure.

Instead of this option you can also use the assembler control $RETCHECK in your source.

Related information

Assembler control **$RETCHECK**

Assembler option: --section-info (-t)

Menu entry

1. Select **Assembler » List File**.

2. Enable the option **Generate list file**.

3. Enable the option **List section summary**.

and/or

1. Select **Assembler » Diagnostics**.

2. Enable the option **Display section summary**.

Command line syntax

--section-info[**=***flag*,...]

-t[*flags*]

You can set the following flags:

+/-console	**c/C**	Display section summary on console
+/-list	**l/L**	List section summary in list file

Default: **--section-info=CL**

Default (without flags): **--section-info=cl**

Description

With this option you tell the assembler to display section information. For each section its memory space, size, total cycle counts and name is listed on stdout and/or in the list file.

The cycle count consists of two parts: the total accumulated count for the section and the total accumulated count for all repeated instructions. In the case of nested loops it is possible that the total supersedes the section total.

Example

To writes the section information to the list file and also display the section information on stdout, enter:

```
as166 --list-file --section-info asm.src
```

Related information

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

Assembler option: --silicon-bug

Menu entry

1. Expand **C/C++ Build** and select **Processor**.

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

   *The **CPU problem bypasses and checks** box shows the available workarounds/checks available for the selected processor.*

3. (Optional) Select **Show all CPU problem bypasses and checks**.

4. Click **Select All** or select one or more individual options.

Command line syntax

--silicon-bug[**=**bug,...]

Description

With this option you specify for which hardware problems the assembler should check. Please refer to Chapter 18, *CPU Problem Bypasses and Checks* for the numbers and descriptions. Silicon bug numbers are specified as a comma separated list. When this option is used without arguments, all silicon bugs are checked.

Example

To check for problems CPU.16 and CPU.18, enter:

```
as166 --silicon-bug=5,6 test.src
```

Related information

Chapter 18, *CPU Problem Bypasses and Checks*

Assembler option: --symbol-scope (-i)

Menu entry

1. Select **Assembler » Symbols**.

2. Enable or disable the option **Set default symbol scope to global**.

Command line syntax

--symbol-scope=*scope*

-i*scope*

You can set the following scope:

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)

Menu entry

-

Command line syntax

`--version`

`-V`

Description

Display version information. The assembler ignores all other options or input files.

Example

```
as166 --version
```

The assembler does not assemble any files but displays the following version information:

```
TASKING VX-toolset for C166: Assembler    vx.yrz Build nnn
Copyright 2004-year Altium BV            Serial# 00000000
```

Related information

-

Assembler option: --warnings-as-errors

Menu entry

1. Select **Global Options**.

2. Enable the option **Treat warnings as errors**.

Command line syntax

`--warnings-as-errors`[`=`*number*`,...`]

Description

If the assembler encounters an error, it stops assembling. When you use this option without arguments, you tell the assembler to treat all warnings as errors. This means that the exit status of the assembler will be non-zero after one or more assembler warnings. As a consequence, the assembler now also stops after encountering a warning.

You can also limit this option to specific warnings by specifying a comma-separated list of warning numbers.

Related information

Assembler option **--no-warnings** (Suppress some or all warnings)

12.5. Linker Options

This section lists all linker options.

Options in Eclipse versus options on the command line

Most command line options have an equivalent option in Eclipse but some options are only available on the command line. Eclipse invokes the linker via the control program. Therefore, it uses the syntax of the control program to pass options and files to the linker. If there is no equivalent option in Eclipse, you can specify a command line option in Eclipse as follows:

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

    The Properties dialog appears.

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

    In the right pane the Settings appear.

3. On the Tool Settings tab, select **Linker » Miscellaneous**.

4. In the **Additional options** field, enter one or more command line options.

    *Because Eclipse uses the control program, Eclipse automatically precedes the option with **-Wl** to pass the option via the control program directly to the linker.*

Be aware that some command line options are not useful in Eclipse or just do not have any effect. For example, the option **--keep-output-files** keeps files after an error occurred. When you specify this option in Eclipse, it will have no effect because Eclipse always removes the output file after an error had occurred.

Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (**-**) character, long option names always begin with two minus (**--**) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

Options can have flags or suboptions. To switch a flag 'on', use a lowercase letter or a **+***longflag*. To switch a flag off, use an uppercase letter or a **-***longflag*. Separate *longflags* with commas. The following two invocations are equivalent:

```
lk166 -mfkl test.obj
lk166 --map-file-format=+files,+link,+locate test.obj
```

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

Linker option: --case-insensitive

Menu entry

1. Select **Linker » Miscellaneous**.

2. Enable the option **Link case insensitive**.

Command line syntax

`--case-insensitive`

Default: case sensitive

Description

With this option you tell the linker not to distinguish between 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 `.obj` file case insensitive.

Related information

Assembler option **--case-insensitive**

Linker option: --chip-output (-c)

Menu entry

1. Select **Linker » Output Format**.

2. Enable the option **Generate Intel Hex format file** and/or **Generate S-records file**.

3. Enable the option **Create file for each memory chip**.

4. Optionally, specify the **Size of addresses**.

   Eclipse always uses the project name as the basename for the output file.

Command line syntax

--chip-output=[*basename*]**:***format*[**:***addr_size*]**,...**

-c[*basename*]**:***format*[**:***addr_size*]**,...**

You can specify the following formats:

IHEX	Intel Hex
SREC	Motorola S-records

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 Eclipse project or, on the command line, the name of the memory device that was emitted with extension `.hex` or `.sre`. Optionally, you can specify a *basename* which prepends the generated file name.

> The linker always outputs a debugging file in ELF/DWARF format and optionally an absolute object file in Intel Hex-format and/or Motorola S-record format.

Example

To generate Intel Hex output files for each defined memory, enter the following on the command line:

```
lk166 --chip-output=myfile:IHEX test1.obj
```

In this case, this generates the file `myfile_`*memname*`.hex`.

Related information

Linker option **--output** (Output file)

Linker option: --define (-D)

Menu entry

1. Select **Linker » Script File**.

    The Defined symbols box shows the symbols that are currently defined.

2. To define a new symbol, click on the **Add** button in the **Defined symbols** box.

3. Type the symbol definition (for example, `demo=1`)

> Use the **Edit** and **Delete** button to change a macro definition or to remove a macro from the list.

Command line syntax

--define=`macro_name`[**=**`macro_definition`]

-D`macro_name`[**=**`macro_definition`]

Description

With this option you can define a macro and specify it to the linker LSL file preprocessor. If you only specify a macro name (no macro definition), the macro expands as '1'.

You can specify as many macros as you like; just use the option **--define** (**-D**) multiple times. If the command line exceeds the limit of the operating system, you can define the macros in an option file which you then must specify to the linker with the option **--option-file** (**-f**) *file*.

The definition can be tested by the preprocessor with `#if`, `#ifdef` and `#ifndef`, for conditional locating.

Example

To define the symbol `__CPU__` which is used in the linker script file `default.lsl` to include the proper processor specific LSL file, enter:

```
lk166 --define=__CPU__=c167 test.obj
```

Related information

Linker option **--option-file** (Specify an option file)

Linker option: --dep-file

Menu entry

Eclipse uses this option in the background to create a file with extension `.d` (one for every input file).

Command line syntax

--dep-file[**=***file*]

Description

With this option you tell the linker to generate dependency lines that can be used in a Makefile. The dependency information will be generated in addition to the normal output file.

By default, the information is written to the file `lk166.d`. When you specify a filename, all dependencies will be combined in the specified file.

Example

```
lk166 --dep-file=test.dep test.obj
```

The linker links the file `test.obj`, which results in the output file `test.elf`, and generates dependency lines in the file `test.dep`.

Related information

Linker option **--make-target** (Target to use in dependencies file)

Linker option: --diag

Menu entry

1. From the **Window** menu, select **Show View » Other » Basic » Problems**.

   The Problems view is added to the current perspective.

2. In the Problems view right-click on a message.

   A popup menu appears.

3. Select **Detailed Diagnostics Info**.

   A dialog box appears with additional information.

Command line syntax

--diag=[*format*:]{**all** | *nr*,...}

You can set the following output formats:

html	HTML output.
rtf	Rich Text Format.
text	ASCII text.

Default format: text

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to stdout (normally your screen) and in the format you specify. You can specify the following formats: **html**, **rtf** or **text** (default). To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas.

With this option the linker does not link/locate any files.

Example

To display an explanation of message number 106, enter:

```
lk166 --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:

```
lk166 --diag=html:all > lkerrors.html
```

Related information

Section 8.10, *Linker Error Messages*

Linker option: --error-file

Menu entry

-

Command line syntax

--error-file[**=***file*]

Description

With this option the linker redirects error messages to a file. If you do not specify a filename, the error file is `lk166.elk`.

Example

To write errors to `errors.elk` instead of `stderr`, enter:

```
lk166 --error-file=errors.elk test.obj
```

Related information

Section 8.10, *Linker Error Messages*

Linker option: --error-limit

Menu entry

1. Select **Linker » Diagnostics**.

2. Enter a value in the **Maximum number of emitted errors** field.

Command line syntax

`--error-limit=`*number*

Default: 42

Description

With this option you tell the linker to only emit the specified maximum number of errors. When 0 (null) is specified, the linker emits all errors. Without this option the maximum number of errors is 42.

Related information

Section 8.10, *Linker Error Messages*

Linker option: --extern (-e)

Menu entry

-

Command line syntax

--extern=symbol**,...**

-esymbol**,...**

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:

```
lk166 mylib.lib
```

Nothing is linked and no output file will be produced, because there are no unresolved symbols when the linker searches through mylib.lib.

```
lk166 --extern=__START mylib.lib
```

In this case the linker searches for the symbol __START in the library and (if found) extracts the object that contains __START, the startup code. If this module contains new unresolved symbols, the linker looks again in mylib.lib. This process repeats until no new unresolved symbols are found.

Related information

Section 8.3, *Linking with Libraries*

Linker option: --first-library-first

Menu entry

-

Command line syntax

`--first-library-first`

Description

When the linker processes a library it searches for symbols that are referenced by the objects and libraries processed so far. If the library contains a definition for an unresolved reference the linker extracts the object that contains the definition from the library.

By default the linker processes object files and libraries in the order in which they appear on the command line. If you specify the option **--first-library-first** the linker always tries to take the symbol definition from the library that appears first on the command line before scanning subsequent libraries.

This is for example useful when you are working with a newer version of a library that partially overlaps the older version. Because they do not contain exactly the same functions, you have to link them both. However, when a function is present in both libraries, you may want the linker to extract the most recent function.

Example

Consider the following example:

```
lk166 --first-library-first a.lib test.obj b.lib
```

If the file `test.obj` calls a function which is both present in `a.lib` and `b.lib`, normally the function in `b.lib` would be extracted. With this option the linker first tries to extract the symbol from the first library `a.lib`.

Note that routines in `b.lib` that call other routines that are present in both `a.lib` and `b.lib` are now also resolved from `a.lib`.

Related information

Linker option **--no-rescan** (Rescan libraries to solve unresolved externals)

Linker option: --global-type-checking

Menu entry

-

Command line syntax

`--global-type-checking`

Description

Use this option when you want the linker to check the types of variable and function references against their definitions, using DWARF 3 debug information.

This check should give the same result as the C compiler when you use MIL linking.

Related information

C compiler option **--global-type-checking** (Global type checking)

Linker option: --help (-?)

Menu entry

-

Command line syntax

--help[**=**_item_]

-?

You can specify the following arguments:

options   Show extended option descriptions

Description

Displays an overview of all command line options. When you specify the argument **options** you can list detailed option descriptions.

Example

The following invocations all display a list of the available command line options:

```
lk166 -?
lk166 --help
lk166
```

To see a detailed description of the available options, enter:

```
lk166 --help=options
```

Related information

-

Linker option: --hex-format

Menu entry

1. Select **Linker » Output Format**.

2. Enable the option **Generate Intel Hex format file**.

3. Enable or disable the option**Emit start address record**.

Command line syntax

```
--hex-format=flag,...
```

You can set the following flag:

    +/-start-address   **s/S**   Emit start address record

Default: `--hex-format=s`

Description

With this option you can specify to emit or omit the start address record from the hex file.

Related information

Linker option **--output** (Output file)

Linker option: --hex-record-size

Menu entry

1. Select **Linker » Output Format**.

2. Enable the option **Generate Intel Hex format file**.

3. Select **Linker » Miscellaneous**.

4. Add the option **--hex-record-size** to the **Additional options** field.

Command line syntax

```
--hex-record-size=size
```

Default: 32

Description

With this option you can set the size (width) of the Intel Hex data records.

Related information

Linker option **--output** (Output file)

Section 15.2, *Intel Hex Record Format*

Linker option: --import-object

Menu entry

1. Select **Linker » Data Objects**.

    The Data objects box shows the list of object files that are imported.

2. To add a data object, click on the **Add** button in the **Data objects** box.

3. Type or select a binary file (including its path).

Use the **Edit** and **Delete** button to change a filename or to remove a data object from the list.

Command line syntax

--import-object=*file*,...

Description

With this option the linker imports a binary *file* containing raw data and places it in a section. The section name is derived from the filename, in which dots are replaced by an underscore. So, when importing a file called `my.jpg`, a section with the name `my_jpg` is created. In your application you can refer to the created section by using linker labels.

Related information

Section 8.5, *Importing Binary Files*

Linker option: --include-directory (-I)

Menu entry

-

Command line syntax

--include-directory=$path$,...

-I$path$,...

Description

With this option you can specify the path where your LSL include files are located. A relative path will be relative to the current directory.

The order in which the linker searches for LSL include files is:

1. The pathname in the LSL file and the directory where the LSL file is located (only for #include files that are enclosed in "")

2. The path that is specified with this option.

3. The default directory `$(PRODDIR)\include.lsl`.

Example

Suppose that your linker script file `mylsl.lsl` contains the following line:

```
#include "myinc.inc"
```

You can call the linker as follows:

```
lk166 --include-directory=c:\proj\include --lsl-file=mylsl.lsl test.obj
```

First the linker looks for the file `myinc.inc` in the directory where `mylsl.lsl` is located. If it does not find the file, it looks in the directory `c:\proj\include` (this option). Finally it looks in the directory `$(PRODDIR)\include.lsl`.

Related information

Linker option **--lsl-file** (Specify linker script file)

Linker option: --incremental (-r)

Menu entry

-

Command line syntax

`--incremental`

`-r`

Description

Normally the linker links and locates the specified object files. With this option you tell the linker only to link the specified files. The linker creates a linker output file `.out`. You then can link this file again with other object files until you have reached the final linker output file that is ready for locating.

In the last pass, you call the linker without this option with the final linker output file `.out`. The linker will now locate the file.

Example

In this example, the files `test1.obj`, `test2.obj` and `test3.obj` are incrementally linked:

1. `lk166 --incremental test1.obj test2.obj --output=test.out`

   test1.obj and test2.obj are linked

2. `lk166 --incremental test3.obj test.out`

   test3.obj and test.out are linked, task1.out is created

3. `lk166 task1.out`

   task1.out is located

Related information

Section 8.4, *Incremental Linking*

Linker option: --keep-output-files (-k)

Menu entry

Eclipse *always* removes the output files when errors occurred.

Command line syntax

```
--keep-output-files
```

```
-k
```

Description

If an error occurs during linking, the resulting output file may be incomplete or incorrect. With this option you keep the generated output files when an error occurs.

By default the linker removes the generated output file when an error occurs. This is useful when you use the make utility. If the erroneous files are not removed, the make utility may process corrupt files on a subsequent invocation.

Use this option when you still want to use the generated file. For example when you know that a particular error does not result in a corrupt object file, or when you want to inspect the output file, or send it to Altium support.

Related information

Linker option **--warnings-as-errors** (Treat warnings as errors)

Linker option: --library (-l)

Menu entry

1. Select **Linker » Libraries**.

   The Libraries box shows the list of libraries that are linked with the project.

2. To add a library, click on the **Add** button in the **Libraries** box.

3. Type or select a library (including its path).

4. Optionally, disable the option **Link default libraries**.

> Use the **Edit** and **Delete** button to change a library name or to remove a library from the list.

Command line syntax

--library=`name`

-l`name`

Description

With this option you tell the linker to use system library c166`name`.lib, where *name* is a string. The linker first searches for system libraries in any directories specified with **--library-directory**, then in the directories specified with the environment variable LIBC166, unless you used the option **--ignore-default-library-path**.

Example

To search in the system library c166cn.lib (C library):

```
lk166 test.obj mylib.lib --library=cn
```

The linker links the file test.obj and first looks in library mylib.lib (in the current directory only), then in the system library c166cn.lib to resolve unresolved symbols.

Related information

Linker option **--library-directory** (Additional search path for system libraries)

Section 8.3, *Linking with Libraries*

Linker option: --library-directory (-L) / --ignore-default-library-path

Menu entry

1. Select **Linker » Libraries**.

    The Library search path box shows the directories that are added to the search path for library files.

2. To define a new directory for the search path, click on the **Add** button in the **Library search path** box.

3. Type or select a path.

Use the **Edit** and **Delete** button to change a path or to remove a path from the list.

Command line syntax

```
--library-directory=path,...
-Lpath,...

--ignore-default-library-path
-L
```

Description

With this option you can specify the path(s) where your system libraries, specified with the option **--library** (**-l**), are located. If you want to specify multiple paths, use the option **--library-directory** for each separate path.

The default path is `$(PRODDIR)\lib`.

If you specify only **-L** (without a pathname) or the long option **--ignore-default-library-path**, the linker will not search the default path and also not in the paths specified in the environment variable `LIBC166`. 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 `LIBC166`.

3. The default directory `$(PRODDIR)\lib`.

Example

Suppose you call the linker as follows:

```
lk166 test.obj --library-directory=c:\mylibs --library=cn
```

First the linker looks in the directory `c:\mylibs` for library `c166cn.lib` (this option). If it does not find the requested libraries, it looks in the directory that is set with the environment variable `LIBC166`. Then the linker looks in the default directory `$(PRODDIR)\lib` for libraries.

Related information

Linker option **--library** (Link system library)

Section 8.3.1, *How the Linker Searches Libraries*

Linker option: --link-only

Menu entry

-

Command line syntax

`--link-only`

Description

With this option you suppress the locating phase. The linker stops after linking and informs you about unresolved references.

Related information

Control program option **--create=relocatable** (**-cl**) (Stop after linking)

Linker option: --lsl-check

Menu entry

-

Command line syntax

`--lsl-check`

Description

With this option the linker just checks the syntax of the LSL file(s) and exits. No linking or locating is performed. Use the option **--lsl-file** to specify the name of the Linker Script File you want to test.

Related information

Linker option **--lsl-file** (Linker script file)

Linker option **--lsl-dump** (Dump LSL info)

Section 8.7, *Controlling the Linker with a Script*

Linker option: --lsl-dump

Menu entry

-

Command line syntax

`--lsl-dump``[`*`=file`*`]`

Description

With this option you tell the linker to dump the LSL part of the map file in a separate file, independent of the option **--map-file** (generate map file). If you do not specify a filename, the file `lk166.ldf` is used.

Related information

Linker option **--map-file-format** (Map file formatting)

Linker option: --lsl-file (-d)

Menu entry

An LSL file can be generated when you create your project in Eclipse:

1.   From the **File** menu, select **File » New » TASKING VX-toolset for C166 C/C++ Project**.

     The New C/C++ Project wizard appears.

2.   Fill in the project settings in each dialog and click **Next >** until the **C166 Project Settings** appear.

3.   Enable the option**Add Linker script file to the project** and click **Finish**.

     Eclipse creates your project and the file project.lsl in the project directory.

The LSL file can be specified in the Properties dialog:

1.   Select **Linker » Script File**.

2.   Specify a LSL file in the **Linker script file (.lsl)** field (default `../${PROJ}.lsl`).

Command line syntax

--lsl-file=`file`

-d`file`

Description

A linker script file contains vital information about the core for the locating phase of the linker. A linker script file is coded in LSL and contains the following types of information:

• the architecture definition describes the core's hardware architecture.

• the memory definition describes the physical memory available in the system.

• the section layout definition describes how to locate sections in memory.

With this option you specify a linker script file to the linker. If you do not specify this option, the linker uses a default script file. You can specify the existing file `default.lsl` or the name of a manually written linker script file. You can use this option multiple times. The linker processes the LSL files in the order in which they appear on the command line.

Related information

Linker option **--lsl-check** (Check LSL file(s) and exit)

Section 8.7, *Controlling the Linker with a Script*

Linker option: --make-target

Menu entry

-

Command line syntax

`--make-target=``name`

Description

With this option you can overrule the default target name in the make dependencies generated by the option **--dep-file**. The default target name is the basename of the input file, with extension `.elf`.

Example

```
lk166 --make-target=mytarget.elf test.obj
```

The linker generates dependency lines with the default target name `mytarget.elf` instead of `test.elf`.

Related information

Linker option **--dep-file** (Generate dependencies in a file)

Linker option: --map-file (-M)

Menu entry

1. Select **Linker » Map File**.

2. Enable the option **Generate XML map file format (.mapxml) for map file viewer**.

3. (Optional) Enable the option **Generate map file**.

4. Enable or disable the types of information to be included.

Command line syntax

--map-file[**=***file*][**:XML**]

-M[*file*][**:XML**]

Default (Eclipse): XML map file is generated

Default (linker): no map file is generated

Description

With this option you tell the linker to generate a linker map file. If you do not specify a filename and you specified the option **--output**, the linker uses the same basename as the output file with the extension .map. If you did not specify the option **--output**, the linker uses the file task1.map. Eclipse names the .map file after the project.

In Eclipse the XML variant of the map file (extension .mapxml) is used for graphical display in the map file viewer.

A linker map file is a text file that shows how the linker has mapped the sections and symbols from the various object files (.obj) to the linked object file. A locate part shows the absolute position of each section. External symbols are listed per space with their absolute address, both sorted on symbol and sorted on address.

Related information

Linker option **--map-file-format** (Format map file)

Section 14.2, *Linker Map File Format*

Linker option: --map-file-format (-m)

Menu entry

1. Select **Linker » Map File**.

2. Enable the option **Generate XML map file format (.mapxml) for map file viewer**.

3. (Optional) Enable the option **Generate map file**.

4. Enable or disable the types of information to be included.

Command line syntax

--map-file-format=$flag$**,...**

-m$flags$

You can set the following flags:

+/-callgraph	**c/C**	Include call graph information
+/-removed	**d/D**	Include information on removed sections
+/-files	**f/F**	Include processed files information
+/-invocation	**i/I**	Include information on invocation and tools
+/-link	**k/K**	Include link result information
+/-locate	**l/L**	Include locate result information
+/-memory	**m/M**	Include memory usage information
+/-nonalloc	**n/N**	Include information of non-alloc sections
+/-overlay	**o/O**	Include overlay information
+/-statics	**q/Q**	Include module local symbols information
+/-crossref	**r/R**	Include cross references information
+/-lsl	**s/S**	Include processor and memory information
+/-rules	**u/U**	Include locate rules

Use the following options for predefined sets of flags:

--map-file-format=0	**-m0**	Link information Alias for **-mcDfikLMNoQrSU**
--map-file-format=1	**-m1**	Locate information Alias for **-mCDfiKlMNoQRSU**
--map-file-format=2	**-m2**	Most information Alias for **-mcdfiklmNoQrSu**

Default: **--map-file-format=2**

Description

With this option you specify which information you want to include in the map file.

On the command line you must use this option in combination with the option **--map-file** (**-M**).

Related information

Linker option **--map-file** (Generate map file)

Section 14.2, *Linker Map File Format*

Linker option: --misra-c-report

Menu entry

-

Command line syntax

`--misra-c-report`[**`=`***`file`*]

Description

With this option you tell the linker to create a MISRA-C Quality Assurance report. This report lists the various modules in the project with the respective MISRA-C settings at the time of compilation. If you do not specify a filename, the file *basename*.`mcr` is used.

Related information

C compiler option **--misrac** (MISRA-C checking)

Linker option: --munch

Menu entry

-

Command line syntax

`--munch`

Description

With this option you tell the linker to activate the muncher in the pre-locate phase.

The muncher phase is a special part of the linker that creates sections containing a list of pointers to the initialization and termination routines. The list of pointers is consulted at run-time by startup code invoked from `_main`, and the routines on the list are invoked at the appropriate times.

Related information

-

Linker option: --non-romable

Menu entry

-

Command line syntax

`--non-romable`

Description

With this option you tell the linker that the application must not be located in ROM. The linker will locate all ROM sections, including a copy table if present, in RAM. When the application is started, the data sections are re-initialized and the BSS sections are cleared as usual.

This option is, for example, useful when you want to test the application in RAM before you put the final application in ROM. This saves you the time of flashing the application in ROM over and over again.

Related information

-

Linker option: --no-rescan

Menu entry

1. Select **Linker » Libraries**.

2. Disable the option **Rescan libraries to solve unresolved externals**.

Command line syntax

```
--no-rescan
```

Description

When the linker processes a library it searches for symbol definitions that are referenced by the objects and libraries processed so far. If the library contains a definition for an unresolved reference the linker extracts the object that contains the definition from the library. The linker processes object files and libraries in the order in which they appear on the command line.

When all objects and libraries are processed the linker checks if there are unresolved symbols left. If so, the default behavior of the linker is to rescan all libraries in the order given at the command line. The linker stops rescanning the libraries when all symbols are resolved, or when the linker could not resolve any symbol(s) during the rescan of all libraries. Notice that resolving one symbol may introduce new unresolved symbols.

With this option, you tell the linker to scan the object files and libraries only once. When the linker has not resolved all symbols after the first scan, it reports which symbols are still unresolved. This option is useful if you are building your own libraries. The libraries are most efficiently organized if the linker needs only one pass to resolve all symbols.

Related information

Linker option **--first-library-first** (Scan libraries in given order)

Linker option: --no-rom-copy (-N)

Menu entry

-

Command line syntax

`--no-rom-copy`

`-N`

Description

With this option the linker will not generate a ROM copy for data sections. A copy table is generated and contains entries to clear BSS sections. However, no entries to copy data sections from ROM to RAM are placed in the copy table.

The data sections are initialized when the application is downloaded. The data sections are not re-initialized when the application is restarted.

Related information

-

Linker option: --no-warnings (-w)

Menu entry

1. Select **Linker » Diagnostics**.

    The Suppress warnings box shows the warnings that are currently suppressed.

2. To suppress a warning, click on the **Add** button in the **Suppress warnings** box.

3. Enter the numbers, separated by commas, of the warnings you want to suppress (for example `135,136`). Or you can use the **Add** button multiple times.

4. To suppress all warnings, enable the option **Suppress all warnings**.

Use the **Edit** and **Delete** button to change a warning number or to remove a number from the list.

Command line syntax

--no-warnings[**=***number*,...]

-w[*number*,...]

Description

With this option you can suppresses all warning messages or specific warning messages.

On the command line this option works as follows:

*   If you do not specify this option, all warnings are reported.

*   If you specify this option but without numbers, all warnings are suppressed.

*   If you specify this option with a number, only the specified warning is suppressed. You can specify the option **--no-warnings=***number* multiple times.

Example

To suppress warnings 135 and 136, enter:

```
lk166 --no-warnings=135,136 test.obj
```

Related information

Linker option **--warnings-as-errors** (Treat warnings as errors)

Linker option: --optimize (-O)

Menu entry

1. Select **Linker » Optimization**.

2. Select one or more of the following options:

   - Delete unreferenced sections

   - Use a 'first-fit decreasing' algorithm

   - Compress copy table

   - Delete duplicate code

   - Delete duplicate data

   - Compress ROM sections of copy table items

Command line syntax

--optimize=*flag*,...

-O*flags*

You can set the following flags:

+/-delete-unreferenced-sections	c/C	Delete unreferenced sections from the output file
+/-first-fit-decreasing	l/L	Use a 'first-fit decreasing' algorithm to locate unrestricted sections in memory
+/-copytable-compression	t/T	Emit smart restrictions to reduce copy table size
+/-delete-duplicate-code	x/X	Delete duplicate code sections from the output file
+/-delete-duplicate-data	y/Y	Delete duplicate constant data from the output file
+/-copytable-item-compression	z/Z	Try to compress ROM sections of copy table items

Use the following options for predefined sets of flags:

--optimize=0	**-O0**	No optimization Alias for **-OCLTXYZ**
--optimize=1	**-O1**	Default optimization Alias for **-OcLtxyZ**
--optimize=2	**-O2**	All optimizations Alias for **-Ocltxyz**

Default: `--optimize=1`

Description

With this option you can control the level of optimization.

Related information

For details about each optimization see Section 8.6, *Linker Optimizations*.

Linker option: --option-file (-f)

Menu entry

1. Select **Linker » Miscellaneous**.

2. Add the option **--option-file** to the **Additional options** field.

    Be aware that the options in the option file are added to the linker options you have set in the other pages. Only in extraordinary cases you may want to use them in combination.

Command line syntax

--option-file=*file*,...

-f *file*,...

Description

This option is primarily intended for command line use. Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the linker.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

Option files can also be generated on the fly, for example by the make utility. You can specify the option **--option-file** multiple times.

Format of an option file

*   Multiple arguments on one line in the option file are allowed.

*   To include whitespace in an argument, surround the argument with single or double quotes.

*   If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

    ```
    "This has a single quote ' embedded"

    'This has a double quote " embedded'

    'This has a double quote " and a single quote '"' embedded"
    ```

*   When a text line reaches its length limit, use a **** to continue the line. Whitespace between quotes is preserved.

    ```
    "This is a continuation \
    line"

            -> "This is a continuation line"
    ```

• It is possible to nest command line files up to 25 levels.

Example

Suppose the file `myoptions` contains the following lines:

```
--map-file=my.map              (generate a map file)
test.obj                       (input file)
--library-directory=c:\mylibs  (additional search path for system libraries)
```

Specify the option file to the linker:

```
lk166 --option-file=myoptions
```

This is equivalent to the following command line:

```
lk166 --map-file=my.map test.obj --library-directory=c:\mylibs
```

Related information

-

Linker option: --output (-o)

Menu entry

1. Select **Linker » Output Format**.

2. Enable one or more output formats.

   For some output formats you can specify a number of suboptions.

   Eclipse always uses the project name as the basename for the output file.

Command line syntax

--output=[*filename*][**:**format[**:**addr_size][**,**space_name]]...

-o[*filename*][**:**format[**:**addr_size][**,**space_name]]...

You can specify the following formats:

ELF	ELF/DWARF
IHEX	Intel Hex
SREC	Motorola S-records

Description

By default, the linker generates an output file in ELF/DWARF format, with the name `task1.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 `near`, enter:

```
lk166 test.obj --output=myfile.hex:IHEX:2,near
```

If they exist, any other address spaces are emitted as well and are named `myfile_`*spacename*`.hex`.

Related information

Linker option **--chip-output** (Generate an output file for each chip)

Linker option **--hex-format** (Specify Hex file format settings)

Linker option: --print-mangled-symbols (-P)

Menu entry

-

Command line syntax

`--print-mangled-symbols`

`-P`

Description

C++ compilers generate unreadable symbol names. These symbols cannot easily be related to your C++ source file anymore. Therefore the linker will by default decode these symbols conform the IA64 ABI when printed to stdout. With this option you can override this default setting and print the mangled names instead.

Related information

-

Linker option: --strip-debug (-S)

Menu entry

1.   Select **Linker » Miscellaneous**.

2.   Enable the option **Strip symbolic debug information**.

Command line syntax

`--strip-debug`

`-S`

Description

With this option you specify not to include symbolic debug information in the resulting output file.

Related information

-

Linker option: --user-provided-initialization-code (-i)

Menu entry

1. Select **Linker » Miscellaneous**.

2. Enable the option **Do not use standard copy table for initialization**.

Command line syntax

`--user-provided-initialization-code`

`-i`

Description

It is possible to use your own initialization code, for example, to save ROM space. With this option you tell the linker *not* to generate a copy table for initialize/clear sections. Use linker labels in your source code to access the positions of the sections when located.

If the linker detects references to the TASKING initialization code, an error is emitted: it is either the TASKING initialization routine or your own, not both.

Note that the options **--no-rom-copy** and **--non-romable**, may vary independently. The 'copytable-compression' optimization (**--optimize=t**) is automatically disabled when you enable this option.

Related information

Linker option **--no-rom-copy** (Do not generate ROM copy)

Linker option **--non-romable** (Application is not romable)

Linker option **--optimize** (Specify optimization)

Linker option: --verbose (-v) / --extra-verbose (-vv)

Menu entry

-

Command line syntax

```
--verbose / --extra-verbose
```

```
-v / -vv
```

Description

With this option you put the linker in *verbose* mode. The linker prints the link phases while it processes the files. In the *extra verbose* mode, the linker also prints the filenames and it shows which objects are extracted from libraries. With this option you can monitor the current status of the linker.

Related information

-

Linker option: --version (-V)

Menu entry

-

Command line syntax

`--version`

`-v`

Description

Display version information. The linker ignores all other options or input files.

Example

```
lk166 --version
```

The linker does not link any files but displays the following version information:

```
TASKING VX-toolset for C166: Linker    vx.yrz Build nnn
Copyright 2004-year Altium BV          Serial# 00000000
```

Related information

-

Linker option: --warnings-as-errors

Menu entry

1. Select **Global Options**.

2. Enable the option **Treat warnings as errors**.

Command line syntax

`--warnings-as-errors`[`=`*number*`,...`]

Description

When the linker detects an error or warning, it tries to continue the link process and reports other errors and warnings. When you use this option without arguments, you tell the linker to treat all warnings as errors. This means that the exit status of the linker will be non-zero after the detection of one or more linker warnings. As a consequence, the linker will not produce any output files.

You can also limit this option to specific warnings by specifying a comma-separated list of warning numbers.

Related information

Linker option **--no-warnings** (Suppress some or all warnings)

12.6. Control Program Options

The control program **cc166** facilitates the invocation of the various components of the C166 toolset from a single command line.

Options in Eclipse versus options on the command line

Eclipse invokes the compiler, assembler and linker via the control program. Therefore, it uses the syntax of the control program to pass options and files to the tools. The control program processes command line options either by itself, or, when the option is unknown to the control program, it looks whether it can pass the option to one of the other tools. However, for directly passing an option to the C++ compiler, C compiler, assembler or linker, it is recommended to use the control program options **--pass-c++**, **--pass-c**, **--pass-assembler**, **--pass-linker**.

See the previous sections for details on the options of the tools.

Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (**-**) character, long option names always begin with two minus (**--**) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

Options can have flags or suboptions. To switch a flag 'on', use a lowercase letter or a **+**_longflag_. To switch a flag off, use an uppercase letter or a **-**_longflag_. Separate _longflags_ with commas. The following two invocations are equivalent:

```
cc166 -Wc-Oac test.c
cc166 --pass-c=--optimize=+coalesce,+cse test.c
```

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

Control program option: --address-size

Menu entry

1. Select **Linker » Output Format**.

2. Enable the option **Generate Intel Hex format file** and/or **Generate S-records file**.

3. Specify the **Size of addresses**.

   Eclipse always uses the project name as the basename for the output file.

Command line syntax

```
--address-size=addr_size
```

Description

If you specify IHEX or SREC with the control option **--format**, you can additionally specify the record length to be emitted in the output files.

With this option you can specify the size of the addresses in bytes (record length). For Intel Hex you can use the values **1**, **2** or **4** bytes (default). For Motorola-S you can specify: **2** (S1 records), **3** (S2 records) or **4** bytes (S3 records, default).

If you do not specify *addr_size*, the default address size is generated.

Example

To create the SREC file `test.sre` with S1 records, type:

```
cc166 --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: --alternative-sfr-file

Menu entry

1. Select **C/C++ Compiler » Preprocessing**.

2. Enable the option **Use alternative SFR file format**.

Command line syntax

```
--alternative-sfr-file
```

Description

With this option the C/C++ compiler includes the alternative SFR file without alias definitions for SFRs and SFR bit-fields. This alternative SFR file is named reg*cpu*.asfr and is located in the product's include/sfr directory.

Use this option to speed up your compilation (smaller SFR file) and have less namespace pollution.

Related information

Section 1.3.5, *Accessing Hardware from C*

Control program option: --automatic-near

Menu entry

1. Select **C/C++ Compiler » Allocation**.

2. Enable the option **Application wide automatic near data allocation**.

Command line syntax

```
--automatic-near
```

Description

With this option you enable automatic near data allocation, where default pointer qualifiers may be replaced by more optimal qualifiers.

This option can only be used together with the MIL linking or MIL splitting build process, because it needs application scope. Also this option can only be used in the memory models far, shuge and huge.

Related information

Section 4.6.2, *Core Specific Optimizations (backend)*

Control program option **--mil-link** / **--mil-split**

Control program option: --bita-struct-threshold

Menu entry

1. Select **C/C++ Compiler » Allocation**

2. In the **Threshold for putting structs in __bita** field, enter a value in bytes.

Command line syntax

`--bita-struct-threshold=`*threshold*

Default: `--bita-struct-threshold=0`

Description

With this option the compiler allocates unqualified structures that are smaller than or equal to the specified *threshold* and have a bit-field of length one in the bit-addressable (`__bita`) memory space automatically. The *threshold* must be specified in bytes. Objects that are qualified `const` or `volatile` or objects that are absolute are not moved.

By default the *threshold* is 0 (off), which means that no objects are moved.

Note that this option has no effect when you use MIL linking/splitting (**--mil-link**/**--mil-split**).

Example

To put all unqualified structures with a size of 4 bytes or smaller and a bit-field of length one into the `__bita` section:

```
cc166 --bita-struct-threshold=4 test.c
```

Related information

Control program option **--mil-link** / **--mil-split**

Control program option: --check

Menu entry

-

Command line syntax

`--check`

Description

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

The compiler/assembler reports any warnings and/or errors.

This option is available on the command line only.

Related information

C compiler option **--check** (Check syntax)

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

Control program option: --core

Menu entry

1. Expand **C/C++ Build** and select **Processor**.

2. From the **Processor selection** list, expand **Custom** and select a core.

Command line syntax

--core=_core_

You can specify the following _core_ arguments:

c16x	C16x instruction set
st10	ST10 instruction set
st10mac	ST10 with MAC co-processor support
c166sv1	C166S v1.0 support
xc16x	XC16x/XE16xx/XC2xxx instruction set
super10	Super10 instruction set
super10m345	Enhanced Super10M345 instruction set

Default: derived from **--cpu** if specified and known or else **--core=xc16x**

Description

With this option you specify the core architecture for a target processor for which you create your application. If **--cpu** is specified and the supplied CPU is known by the control program, the control program selects the correct core automatically.

For more information see control program option **--cpu**.

Example

Specify a custom core:

```
cc166 --core=st10 test.c
```

Related information

Control program option **--cpu** (Select processor)

Control program option **--processors** (Read additional processor definitions)

Control program option: --cpu (-C)

Menu entry

1. Expand **C/C++ Build** and select **Processor**.

2. From the **Processor selection** list, select a processor or expand **Custom** and select a core.

Command line syntax

--cpu=_cpu_

-C_cpu_

Description

With this option you define the target processor for which you create your application. The _cpu_ can be a full processor name, like XC2287-72F, or a base CPU name, like xc2287.

Based on this option the C++ compiler and C compiler always include the special function register file reg_cpu_.sfr, unless you specify option **--no-tasking-sfr**.

For C/C++ files, the control program will not pass the **--cpu** option to the assembler, but only the **--core** option. The reason for this is that in general all SFRs are already processed by the C compiler, and there are no SFRs left in the assembly code. This speeds up the build time, because the assembler does not need to read the SFR file. When you use in-line assembly in your C code that uses SFRs, you do want the assembler to read the SFR file. In this case you need to pass the **--cpu** option directly to the assembler (**--pass-assembler=--cpu=**_cpu_ / **-Wa-C**_cpu_). It is recommended to use this option only for those modules that use in-line assembly. This avoids the impact to the build speed to a minimum.

Note that for .asm and .src files the control program will pass the **--cpu** option to the assembler.

If you select a target from the list, the core is known. If you specify a Custom processor, you need to select the core that matches the core of your custom processor (option **--core**). The tools know all CPUs with core c16x, st10, st10mac and super10.

The following table show the relation between the two options:

--cpu=_cpu_	--core=_core_	_core_	Register file
no	no	c16x	c166, cp166: none as166: regc16x.sfr
no	yes	_core_	c166, cp166: none as166: reg_core_.sfr
yes	no	derived from _cpu_ for core c16x, st10, st10mac and super10	c166, cp166: reg_cpu_.sfr as166: reg_core_.sfr
yes	yes	_core_	c166, cp166: reg_cpu_.sfr as166: reg_core_.sfr

The standard list of supported processors is defined in the file processors.xml. This file defines for each processor its full name (for example, XC2287-72F), the base CPU name (for example, xc2287), the

core settings (for example, xc16x), the on-chip flash settings, the list of silicon bugs for that processor. Each processor also defines an option to supply to the linker for preprocessing the LSL file for the applicable on-chip memory definitions. The option is for example **-DXC2287_72M**.

The preferred use of the option **--cpu**, is to specify the full processor name. For example, **--cpu=XC2287-72F**. The control program will lookup this processor name in the file `processors.xml`. The control program passes the options to the underlaying tools. For example, **--cpu=XC2287-72F --core=xc16x** to the C compiler, or **-D__CPU__=xc2287 -DXC2287_72M** to the linker.

The control program also supports the use of the base CPU name as argument of **--cpu**. In that case the control program will lookup the first processor in the file `processors.xml` that has this base CPU. If multiple processors exist with the same base CPU, a warning will be issued and the first is selected:

```
cc166 W752: the name 'xc2287' specified with -C or --cpu is not a full
processor name and is not a unique base CPU name, selected 'XC2287-56F'
```

In practice the differences between processors with the same base CPU is a difference in on-chip memories.

Example

Specify an existing processor:

```
cc166 --cpu=XC2287-72F test.c
```

Specify a custom processor:

```
cc166 --cpu=custom --core=st10 test.c
```

Related information

Control program option **--core** (Select the core)

Control program option **--processors** (Read additional processor definitions)

Control program option **--no-tasking-sfr** (Do not include SFR file)

Control program option: --create (-c)

Menu entry

-

Command line syntax

--create[**=**stage]

-c[stage]

You can specify the following stages:

intermediate-c	**c**	Stop after C++ files are compiled to intermediate C files (.ic)
relocatable	**l**	Stop after the files are linked to a linker object file (.out)
mil	**m**	Stop after C++ files or C files are compiled to MIL (.mil)
object	**o**	Stop after the files are assembled to objects (.obj)
assembly	**s**	Stop after C++ files or C files are compiled to assembly (.src)

Default (without flags): **--create=object**

Description

Normally the control program generates an absolute object file of the specified output format from the file you supplied as input. With this option you tell the control program to stop after a certain number of phases.

Example

To generate the object file test.obj:

cc166 --create test.c

The control program stops after the file is assembled. It does not link nor locate the generated output.

Related information

Linker option **--link-only** (Link only, no locating)

Control program option: --debug-info (-g)

Menu entry

1. Select **C/C++ Compiler » Debugging**.

2. To **generate symbolic debug information**, select **Default**, **Small set** or **Full**.
    To disable the generation of debug information, select **None**.

Command line syntax

```
--debug-info
```

```
-g
```

Description

With this option you tell the control program to include debug information in the generated object file.

The control program passes the option **--debug-info** (**-g**) to the C compiler and calls the assembler with **--debug-info=+smart,+local** (**-gsl**).

Related information

C compiler option **--debug-info** (Generate symbolic debug information)

Assembler option **--debug-info** (Generate symbolic debug information)

Control program option: --define (-D)

Menu entry

1. Select **C/C++ Compiler » Preprocessing** and/or **Assembler » Preprocessing**.

    The Defined symbols box right-below shows the symbols that are currently defined.

2. To define a new symbol, click on the **Add** button in the **Defined symbols** box.

3. Type the symbol definition (for example, `demo=1`)

Use the **Edit** and **Delete** button to change a macro definition or to remove a macro from the list.

Command line syntax

--define=`macro_name`[**=**`macro_definition`]

-D`macro_name`[**=**`macro_definition`]

Description

With this option you can define a macro and specify it to the preprocessor. If you only specify a macro name (no macro definition), the macro expands as '1'.

You can specify as many macros as you like. Simply use the **Add** button to add new macro definitions.

On the command line, use the option **--define** (**-D**) multiple times. If the command line exceeds the limit of the operating system, you can define the macros in an option file which you then must specify to the compiler with the option **--option-file** (**-f**) *file*.

Defining macros with this option (instead of in the C source) is, for example, useful to compile conditional C source as shown in the example below.

The control program passes the option **--define** (**-D**) to the compiler and the assembler.

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:

```
cc166 --define=DEMO test.c
cc166 --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.

```
cc166 --define="MAX(A,B)=((A) > (B) ? (A) : (B))" test.c
```

Related information

Control program option **--undefine** (Remove preprocessor macro)

Control program option **--option-file** (Specify an option file)

Control program option: --dep-file

Menu entry

-

Command line syntax

--dep-file[**=**_file_]

Description

With this option you tell the compiler to generate dependency lines that can be used in a Makefile. In contrast to the option **--preprocess=+make**, the dependency information will be generated in addition to the normal output file.

By default, the information is written to a file with extension .d (one for every input file). When you specify a filename, all dependencies will be combined in the specified file.

Example

```
cc166 --dep-file=test.dep -t test.c
```

The compiler compiles the file test.c, which results in the output file test.src, and generates dependency lines in the file test.dep.

Related information

Control program option **--preprocess=+make** (Generate dependencies for make)

Control program option: --diag

Menu entry

1.   From the **Window** menu, select **Show View » Other » General » Problems**.

      The Problems view is added to the current perspective.

2.   In the Problems view right-click on a message.

      A popup menu appears.

3.   Select **Detailed Diagnostics Info**.

      A dialog box appears with additional information.

Command line syntax

--diag=[*format*:]{**all** | *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:

```
cc166 --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:

```
cc166 --diag=html:all > ccerrors.html
```

Related information

Section 4.9, *C Compiler Error Messages*

Control program option: --dry-run (-n)

Menu entry

-

Command line syntax

`--dry-run`

`-n`

Description

With this option you put the control program in verbose mode. The control program prints the invocations of the tools it would use to process the files without actually performing the steps.

Related information

Control program option **--verbose** (Verbose output)

Control program option: --error-file

Menu entry

-

Command line syntax

`--error-file`

Description

With this option the control program tells the compiler, assembler and linker to redirect error messages to a file.

Example

To write errors to error files instead of stderr, enter:

```
cc166 --error-file test.c
```

Related information

Control Program option **--warnings-as-errors** (Treat warnings as errors)

Control program option: --exceptions

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Support for C++ exception handling**.

Command line syntax

`--exceptions`

Description

With this option you enable support for exception handling in the C++ compiler.

Related information

-

Control program option: --force-c

Menu entry

-

Command line syntax

`--force-c`

Description

With this option you tell the control program to treat all `.cc` files as C files instead of C++ files. This means that the control program does not call the C++ compiler and forces the linker to link C libraries.

Related information

Control program option **--force-c++** (Force C++ compilation and linking)

Control program option: --force-c++

Menu entry

Eclipse always uses this option for a C++ project.

Command line syntax

```
--force-c++
```

Description

With this option you tell the control program to treat all `.c` files as C++ files instead of C files. This means that the control program calls the C++ compiler prior to the C compiler and forces the linker to link C++ libraries.

Related information

Control program option **--force-c** (Treat C++ files as C files)

Control program option: --force-munch

Menu entry

Eclipse always uses this option for a C++ project.

Command line syntax

`--force-munch`

Description

With this option you force the control program to activate the muncher in the pre-locate phase.

Related information

-

Control program option: --format

Menu entry

1. Select **Linker » Output Format**.

2. Enable the option **Generate Intel Hex format file** and/or **Generate S-records file**.

3. Optionally, specify the **Size of addresses**.

   Eclipse always uses the project name as the basename for the output file.

Command line syntax

--format=_format_

You can specify the following formats:

ELF	ELF/DWARF
IHEX	Intel Hex
SREC	Motorola S-records

Description

With this option you specify the output format for the resulting (absolute) object file. The default output format is ELF/DWARF, which can directly be used by the debugger.

If you choose IHEX or SREC, you can additionally specify the address size of the chosen format (option **--address-size**).

Example

To generate a Motorola S-record output file:

```
cc166 --format=SREC test1.c test2.c --output=test.sre
```

Related information

Control program option **--address-size** (Set address size for linker IHEX/SREC files)

Control program option **--output** (Output file)

Linker option **--chip-output** (Generate an output file for each chip)

Control program option: --fp-trap

Menu entry

1. Select **Linker » Libraries**.

2. Enable the option **Use trapped floating-point library**.

Command line syntax

```
--fp-trap
```

Description

By default the control program uses the non-trapping floating-point library (`c166fpn.lib`). With this option you tell the control program to use the trapping floating-point library (`c166fpnt.lib`).

If you use the trapping floating-point library, exceptional floating-point cases are intercepted and can be handled separately by an application defined exception handler. Using this library decreases the execution speed of your application.

Related information

Section 8.3, *Linking with Libraries*

Control program option: --global-type-checking

Menu entry

1. Select **C/C++ Compiler » Diagnostics**.

2. Enable the option **Perform global type checking on C code**.

Command line syntax

```
--global-type-checking
```

Description

The C compiler already performs type checking within each module. Use this option when you want the linker to perform type checking between modules. The control program passes this option to both the C compiler and the linker.

Related information

-

Control program option: --help (-?)

Menu entry

-

Command line syntax

--help[**=***item*]

-?

You can specify the following argument:

options   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:

```
cc166 -?
cc166 --help
cc166
```

To see a detailed description of the available options, enter:

```
cc166 --help=options
```

Related information

-

Control program option: --include-directory (-I)

Menu entry

1.   Select **C/C++ Compiler » Include Paths**.

     The Include paths box shows the directories that are added to the search path for include files.

2.   To define a new directory for the search path, click on the **Add** button in the **Include paths** box.

3.   Type or select a path.

Use the **Edit** and **Delete** button to change a path or to remove a path from the list.

Command line syntax

`--include-directory=`*path*`,...`

`-I`*path*`,...`

Description

With this option you can specify the path where your include files are located. A relative path will be relative to the current directory.

The control program passes this option to the compiler and the assembler.

Example

Suppose that the C source file `test.c` contains the following lines:

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

You can call the control program as follows:

```
cc166 --include-directory=myinclude test.c
```

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

The compiler now looks for the file `myinc.h` in the directory where `test.c` is located. If the file is not there the compiler searches in the directory `myinclude`. If it was still not found, the compiler searches in the environment variable and then in the default include directory.

Related information

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

C compiler option **--include-file** (Include file at the start of a compilation)

Control program option: --instantiate

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Select an instantiation mode in the **Instantiation mode of external template entities** box.

Command line syntax

--instantiate=_mode_

You can specify the following modes:

> **used**
>
> **all**
>
> **local**

Default: **--instantiate=used**

Description

Control instantiation of external template entities. External template entities are external (that is, non-inline and non-static) template functions and template static data members. The instantiation mode determines the template entities for which code should be generated based on the template definition. Normally, when a file is compiled, template entities are instantiated wherever they are used (the linker will discard duplicate definitions). The overall instantiation mode can, however, be changed with this option. You can specify the following modes:

used	Instantiate those template entities that were used in the compilation. This will include all static data members for which there are template definitions. This is the default.
all	Instantiate all template entities declared or referenced in the compilation unit. For each fully instantiated template class, all of its member functions and static data members will be instantiated whether or not they were used. Non-member template functions will be instantiated even if the only reference was a declaration.
local	Similar to **--instantiate=used** except that the functions are given internal linkage. This is intended to provide a very simple mechanism for those getting started with templates. The compiler will instantiate the functions that are used in each compilation unit as local functions, and the program will link and run correctly (barring problems due to multiple copies of local static variables). However, one may end up with many copies of the instantiated functions, so this is not suitable for production use.

You cannot use **--instantiate=local** in conjunction with automatic template instantiation.

Related information

Control program option **--no-auto-instantiation** (Disable automatic C++ instantiation)

Section 2.5, *Template Instantiation*

Control program option: --integer-enumeration

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Treat enumerated types always as integer**.

Command line syntax

`--integer-enumeration`

Description

Normally the compiler treats enumerated types as the smallest data type possible (`char` instead of `int`). This reduces code size. With this option the compiler always treats enum-types as `int` as defined in the ISO C99 standard.

Related information

Section 1.1, *Data Types*

Control program option: --io-streams

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Support for C++ I/O streams**.

Command line syntax

`--io-streams`

Description

As I/O streams require substantial resources they are disabled by default. Use this option to enable I/O streams support in the C++ library.

This option also enables exception handling.

Related information

-

Control program option: --iso

Menu entry

1. Select **C/C++ Compiler » Language**.

2. From the **Comply to C standard** list, select **ISO C99** or **ISO C90**.

Command line syntax

`--iso={90|99}`

Default: `--iso=99`

Description

With this option you select the ISO C standard. C90 is also referred to as the "ANSI C standard". C99 refers to the newer ISO/IEC 9899:1999 (E) standard. C99 is the default.

Independent of the chosen ISO standard, the control program always links libraries with C99 support.

Example

To select the ISO C90 standard on the command line:

```
cc166 --iso=90 test.c
```

Related information

C compiler option **--iso** (ISO C standard)

Control program option: --keep-output-files (-k)

Menu entry

Eclipse *always* removes generated output files when an error occurs.

Command line syntax

`--keep-output-files`

`-k`

Description

If an error occurs during the compilation, assembling or linking process, the resulting output file may be incomplete or incorrect. With this option you keep the generated output files when an error occurs.

By default the control program removes generated output files when an error occurs. This is useful when you use the make utility. If the erroneous files are not removed, the make utility may process corrupt files on a subsequent invocation.

Use this option when you still want to use the generated files. For example when you know that a particular error does not result in a corrupt file, or when you want to inspect the output file, or send it to Altium support.

The control program passes this option to the compiler, assembler and linker.

Example

```
cc166 --keep-output-files test.c
```

When an error occurs during compiling, assembling or linking, the erroneous generated output files will not be removed.

Related information

C compiler option **--keep-output-files**

Assembler option **--keep-output-files**

Linker option **--keep-output-files**

Control program option: --keep-temporary-files (-t)

Menu entry

1. Select **Global Options**.

2. Enable the option **Keep temporary files**.

Command line syntax

`--keep-temporary-files`

`-t`

Description

By default, the control program removes intermediate files like the `.src` file (result of the compiler phase) and the `.obj` file (result of the assembler phase).

With this option you tell the control program to keep temporary files it generates during the creation of the absolute object file.

Example

```
cc166 --keep-temporary-files test.c
```

The control program keeps all intermediate files it generates while creating the absolute object file `test.elf`.

Related information

-

Control program option: --library (-l)

Menu entry

1. Select **Linker » Libraries**.

    The Libraries box shows the list of libraries that are linked with the project.

2. To add a library, click on the **Add** button in the **Libraries** box.

3. Type or select a library (including its path).

4. Optionally, disable the option **Link default libraries**.

> Use the **Edit** and **Delete** button to change a library name or to remove a library from the list.

Command line syntax

--library=*name*

-l*name*

Description

With this option you tell the linker via the control program to use system library c166*name*.lib, where *name* is a string. The linker first searches for system libraries in any directories specified with **--library-directory**, then in the directories specified with the environment variable LIBC166, unless you used the option **--ignore-default-library-path**.

Example

To search in the system library c166cn.lib (C library):

```
cc166 test.obj mylib.lib --library=cn
```

The linker links the file test.obj and first looks in library mylib.lib (in the current directory only), then in the system library c166cn.lib to resolve unresolved symbols.

Related information

Control program option **--no-default-libraries** (Do not link default libraries)

Control program option **--library-directory** (Additional search path for system libraries)

Section 8.3, *Linking with Libraries*

Control program option: --library-directory (-L) / --ignore-default-library-path

Menu entry

1. Select **Linker » Libraries**.

    The Library search path box shows the directories that are added to the search path for library files.

2. To define a new directory for the search path, click on the **Add** button in the **Library search path** box.

3. Type or select a path.

    Use the **Edit** and **Delete** button to change a path or to remove a path from the list.

Command line syntax

```
--library-directory=path,...
-Lpath,...

--ignore-default-library-path
-L
```

Description

With this option you can specify the path(s) where your system libraries, specified with the option **--library** (**-l**), are located. If you want to specify multiple paths, use the option **--library-directory** for each separate path.

The default path is `$(PRODDIR)\lib`.

If you specify only **-L** (without a pathname) or the long option **--ignore-default-library-path**, the linker will not search the default path and also not in the paths specified in the environment variable `LIBC166`. 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 `LIBC166`.

3. The default directory `$(PRODDIR)\lib`.

Example

Suppose you call the control program as follows:

```
cc166 test.c --library-directory=c:\mylibs --library=cn
```

First the linker looks in the directory `c:\mylibs` for library `c166cn.lib` (this option). If it does not find the requested libraries, it looks in the directory that is set with the environment variable `LIBC166`. Then the linker looks in the default directory `$(PRODDIR)\lib` for libraries.

Related information

Control program option **--library** (Link system library)

Section 8.3.1, *How the Linker Searches Libraries*

Control program option: --list-files

Menu entry

-

Command line syntax

--list-files[**=**_file_]

Default: no list files are generated

Description

With this option you tell the assembler via the control program to generate a list file for each specified input file. A list file shows the generated object code and the relative addresses. Note that the assembler generates a relocatable object file with relative addresses.

With the optional _file_ you can specify a name for the list file. This is only possible if you specify only one input file to the control program. If you do not specify a file name, or you specify more than one input file, the control program names the generated list file(s) after the specified input file(s) with extension `.lst`.

Note that object files and library files are not counted as input files.

Related information

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

Assembler option **--list-format** (Format list file)

Control program option: --lsl-file (-d)

Menu entry

An LSL file can be generated when you create your project in Eclipse:

1. From the **File** menu, select **File » New » TASKING VX-toolset for C166 C/C++ Project**.

   The New C/C++ Project wizard appears.

2. Fill in the project settings in each dialog and click **Next >** until the **C166 Project Settings** appear.

3. Enable the option **Add Linker script file to the project** and click **Finish**.

   Eclipse creates your project and the file project.lsl in the project directory.

The LSL file can be specified in the Properties dialog:

1. Select **Linker » Script File**.

2. Specify a LSL file in the **Linker script file (.lsl)** field (default `../${PROJ}.lsl`).

Command line syntax

--lsl-file=*file*,...

-d*file*,...

Description

A linker script file contains vital information about the core for the locating phase of the linker and for the automatic near data allocation of the C compiler. 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 one or more linker script files via the control program to the linker and the C compiler. If you do not specify this option, the linker uses a default script file. You can specify the existing file `default.lsl` or the name of a manually written linker script file. You can use this option multiple times. The linker processes the LSL files in the order in which they appear on the command line.

Related information

Section 8.7, *Controlling the Linker with a Script*

Control program option: --m166

Menu entry

1. Select **Assembler » Preprocessing**.

2. Enable the option **Use classic macro preprocessor (m166)**.

3. To add controls click on the **Add** button in the **Classic m166 controls** box.

4. Enter a classic control

   .

> Use the **Edit** and **Delete** button to change a control or to remove a control from the list.

Command line syntax

--m166[**=***option*]

Description

With this option you tell the control program to call the classic **m166** preprocessor for assembly sources
(.asm). The *option* argument is passed to **m166** as option or control. You can specify multiple
options/controls to **m166** by using multiple **--m166** options.

This option is useful for backwards compatibility with the classic C166 toolset.

Related information

See the documentation that came with your classic C166 toolset, for more information about the options
and controls of the **m166** macro preprocessor.

Control program option: --make-target

Menu entry

-

Command line syntax

`--make-target=`_name_

Description

With this option you can overrule the default target name in the make dependencies generated by the options **--preprocess=+make** (**-Em**) and **--dep-file**. The default target name is the basename of the input file, with extension `.obj`.

Example

```
cc166 --preprocess=+make --make-target=../mytarget.obj test.c
```

The compiler generates dependency lines with the default target name `../mytarget.obj` instead of `test.obj`.

Related information

Control program option **--preprocess=+make** (Generate dependencies for make)

Control program option **--dep-file** (Generate dependencies in a file)

Control program option: --mil-link / --mil-split

Menu entry

1. Select **C/C++ Compiler » Optimization**.

2. Enable the option **Build for application wide optimizations (MIL linking)**.

3. Select **Optimize less/Build faster** or **Optimize more/Build slower**.

Command line syntax

```
--mil-link
--mil-split[=file,...]
```

Description

With option **--mil-link** the C compiler links the optimized intermediate representation (MIL) of all input files and MIL libraries specified on the command line in the compiler. The result is one single module that is optimized another time.

Option **--mil-split** does the same as option **--mil-link**, but in addition, the resulting MIL representation is written to a file with the suffix .mil and the C compiler also splits the MIL representation and writes it to separate files with suffix .ms. One file is written for each input file or MIL library specified on the command line. The .ms files are only updated on a change.

With option **--mil-split** you can perform application-wide optimizations during the frontend phase by specifying all modules at once, and still invoke the backend phase one module at a time to reduce the total compilation time. Application wide code compaction is not possible in this case.

Optionally, you can specify another filename for the .ms file the C compiler generates. Without an argument, the basename of the C source file is used to create the .ms filename. Note that if you specify a filename, you have to specify one filename for every input file.

Build for application wide optimizations (MIL linking) and Optimize less/Build faster

This option is standard MIL linking and splitting. Note that you can control the optimizations to be performed with the optimization settings.

Optimize more/Build slower

When you enable this option, the compiler's frontend does not split the MIL stream in separate modules, but feeds it directly to the compiler's backend, allowing the code compaction to be performed application wide.

Related information

Section 4.1, *Compilation Process*

C compiler option **--mil** / **--mil-split**

Control program option: --model (-M)

Menu entry

1. Select **C/C++ Compiler » Memory Model**.

2. In the Default data box, select a memory model.

Command line syntax

--model={**near** | **far** | **shuge** | **huge**}

-M{**n** | **f** | **s** | **h**}

Default: **--model=near**

Description

By default, the C166 compiler uses the near memory model. With this memory model the most efficient code is generated. You can specify the option **--model** to specify another memory model.

The table below illustrates the meaning of each memory model:

Model	Memory type	Location	Pointer size	Pointer arithmetic
near	__near	Near data pages defined at link time	16 bit	16 bit
far	__far	Anywhere in memory	32 bit	14 bit
segmented huge	__shuge	Anywhere in memory	32 bit	16 bit
huge	__huge	Anywhere in memory	32 bit	32 bit

The value of the predefined preprocessor symbol __MODEL__ represents the memory model selected with this option. This can be very helpful in making conditional C code in one source module, used for different applications in different memory models. The value of __MODEL__ is:

near model	'n'
far model	'f'
segmented huge model	's'
huge model	'h'

Example

To compile the file test.c for the far memory model:

```
cc166 --model=far test.c
```

Related information

Section 1.3.2, *Memory Models*

Control program option: --near-functions

Menu entry

1. Select **C/C++ Compiler » Memory Model**.

2. Enable the option **Make unqualified functions near**.

Command line syntax

```
--near-functions
```

Description

With this option you tell the compiler to treat unqualified functions as `__near` functions instead of `__huge` functions.

Related information

-

Control program option: --near-threshold

Menu entry

1. Select **C/C++ Compiler » Allocation**

2. In the **Threshold for putting data in __near** field, enter a value in bytes.

Command line syntax

--near-threshold=*threshold*

Default: **--near-threshold=0**

Description

With this option the compiler allocates unqualified objects that are smaller than or equal to the specified *threshold* in the __near memory space automatically. The *threshold* must be specified in bytes. Objects that are qualified const or volatile, external objects or objects that are absolute (__at()) are not moved.

By default the *threshold* is 0 (off), which means that all data is allocated in the default memory space.

You cannot use this option in the near (**--model=near**) memory model.

Example

To put all data objects with a size of 4 bytes or smaller in __near memory:

```
cc166 --model=far --near-threshold=4 test.c
```

Example part of test.c:

```
extern int x;   //will be put in default far memory
int y;          //will be put in near memory
```

Related information

Control program option **--model** (Select memory model)

Control program option: --no-auto-instantiation

Menu entry

-

Command line syntax

`--no-auto-instantiation`

Default: the C++ compiler automatically instantiates templates.

Description

With this option automatic instantiation of templates is disabled.

Related information

Control program option **--instantiate** (Set instantiation mode)

Section 2.5, *Template Instantiation*

Control program option: --no-default-libraries

Menu entry

1. Select **Linker » Libraries**.

2. Disable the option **Link default libraries**.

Command line syntax

```
--no-default-libraries
```

Description

By default the control program specifies the standard C libraries (C99) and run-time library to the linker. With this option you tell the control program not to specify the standard C libraries and run-time library to the linker.

In this case you must specify the libraries you want to link to the linker with the option **--library=**library_name or pass the libraries as files on the command line. The control program recognizes the option **--library** (**-l**) as an option for the linker and passes it as such.

Example

```
cc166 --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 (c166cn.lib) and avoid unresolved externals:

```
cc166 --no-default-libraries --library=cn test.c
```

Related information

Control program option **--library** (Link system library)

Section 8.3.1, *How the Linker Searches Libraries*

Control program option: --no-double (-F)

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Treat double as float**.

Command line syntax

`--no-double`

`-F`

Description

With this option you tell the compiler to treat variables of the type `double` as `float`. Because the float type takes less space, execution speed increases and code size decreases, both at the cost of less precision.

The control program also tells the linker to link the single-precision C library.

Related information

-

Control program option: --no-map-file

Menu entry

1. Select **Linker » Map File**.

2. Disable the option **Generate map file**.

Command line syntax

`--no-map-file`

Description

By default the control program tells the linker to generate a linker map file.

A linker map file is a text file that shows how the linker has mapped the sections and symbols from the various object files (`.obj`) to the linked object file. A locate part shows the absolute position of each section. External symbols are listed per space with their absolute address, both sorted on symbol and sorted on address.

With this option you prevent the generation of a map file.

Related information

-

Control program option: --no-tasking-sfr

Menu entry

1. Select **C/C++ Compiler » Preprocessing**.

2. Disable the option **Automatic inclusion of '.sfr' file**.

3. Select **Assembler » Preprocessing**.

4. Disable the option **Automatic inclusion of '.sfr' file**.

Command line syntax

```
--no-tasking-sfr
```

Description

Normally, the C compiler and assembler includes a special function register (SFR) file before compiling/assembling. The compiler and assembler automatically select the SFR file belonging to the target you selected on the **Processor** page (control program option **--cpu**).

With this option the compiler and assembler do not include the register file reg*cpu*.sfr as based on the selected target processor.

Use this option if you want to use your own set of SFR files.

Related information

Control program option **--cpu** (Select processor)

Section 1.3.5, *Accessing Hardware from C*

Control program option: --no-warnings (-w)

Menu entry

1. Select **C/C++ Compiler » Diagnostics**.

    The Suppress C compiler warnings box shows the warnings that are currently suppressed.

2. To suppress a warning, click on the **Add** button in the **Suppress warnings** box.

3. Enter the numbers, separated by commas or as a range, of the warnings you want to suppress (for example `537,538`). Or you can use the **Add** button multiple times.

4. To suppress all warnings, enable the option **Suppress all warnings**.

> Use the **Edit** and **Delete** button to change a warning number or to remove a number from the list.

Command line syntax

--no-warnings[**=**number[-number],...]

-w[number[-number],...]

Description

With this option you can suppresses all warning messages for the various tools or specific control program warning messages.

On the command line this option works as follows:

- If you do not specify this option, all warnings are reported.

- If you specify this option but without numbers, all warnings of all tools are suppressed.

- If you specify this option with a number or a range, only the specified control program warnings are suppressed. You can specify the option **--no-warnings=**number multiple times.

Example

To suppress all warnings for all tools, enter:

```
cc166 test.c --no-warnings
```

Related information

Control program option **--warnings-as-errors** (Treat warnings as errors)

Control program option: --option-file (-f)

Menu entry

-

Command line syntax

--option-file=_file_,...

-f _file_,...

Description

This option is primarily intended for command line use. Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the control program.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option **--option-file** multiple times.

Format of an option file

- Multiple arguments on one line in the option file are allowed.

- To include whitespace in an argument, surround the argument with single or double quotes.

- If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

  ```
  "This has a single quote ' embedded"

  'This has a double quote " embedded'

  'This has a double quote " and a single quote '"' embedded"
  ```

- When a text line reaches its length limit, use a **** to continue the line. Whitespace between quotes is preserved.

  ```
  "This is a continuation \
  line"

          -> "This is a continuation line"
  ```

- It is possible to nest command line files up to 25 levels.

Example

Suppose the file `myoptions` contains the following lines:

```
--debug-info
--define=DEMO=1
test.c
```

Specify the option file to the control program:

```
cc166 --option-file=myoptions
```

This is equivalent to the following command line:

```
cc166 —-debug-info --define=DEMO=1 test.c
```

Related information

-

Control program option: --output (-o)

Menu entry

Eclipse always uses the project name as the basename for the output file.

Command line syntax

`--output=`*file*

`-o` *file*

Description

By default, the control program generates a file with the same basename as the first specified input file. With this option you specify another name for the resulting absolute object file.

The default output format is ELF/DWARF, but you can specify another output format with option **--format**.

Example

```
cc166 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`:

```
cc166 --output=result.elf test.c prog.c
```

Related information

Control program option **--format** (Set linker output format)

Linker option **--output** (Output file)

Linker option **--chip-output** (Generate an output file for each chip)

Control program option: --pass (-W)

Menu entry

1. Select **C/C++ Compiler » Miscellaneous** or **Assembler » Miscellaneous** or **Linker » Miscellaneous**.

2. Add an option to the **Additional options** field.

    *Be aware that the options in the option file are added to the options you have set in the other pages. Only in extraordinary cases you may want to use them in combination. The assembler options are preceded by **-Wa** and the linker options are preceded by **-Wl**. For the C/C++ options you have to do this manually.*

Command line syntax

--pass-assembler=option	**-Wa**option	Pass option directly to the assembler
--pass-c=option	**-Wc**option	Pass option directly to the C compiler
--pass-c++=option	**-Wcp**option	Pass option directly to the C++ compiler
--pass-linker=option	**-Wl**option	Pass option directly to the linker

Description

With this option you tell the control program to call a tool with the specified option. The control program does not use or interpret the option itself, but specifies it directly to the tool which it calls.

Example

To pass the option **--verbose** directly to the linker, enter:

```
cc166 --pass-linker=--verbose test.c
```

Related information

-

Control program option: --preprocess (-E) / --no-preprocessing-only

Menu entry

1. Select **C/C++ Compiler » Preprocessing**.

2. Enable the option **Store preprocessor output in <file>.pre**.

3. (Optional) Enable the option **Keep comments in preprocessor output**.

4. (Optional) Enable the option **Keep #line info in preprocessor output**.

Command line syntax

--preprocess[**=***flags*]

-E[*flags*]

--no-preprocessing-only

You can set the following flags:

+/-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). Eclipse also compiles the C source.

On the command line, the control program stops after preprocessing. If you also want to compile the C source you can specify the option **--no-preprocessing-only**. In this case the control program calls the compiler twice, once with option **--preprocess** and once for a regular compilation.

With **--preprocess=+comments** you tell the preprocessor to keep the comments from the C source file in the preprocessed output.

With **--preprocess=+includes** the compiler will generate a list of all included source files. The preprocessor output is discarded.

With **--preprocess=+list** the compiler will generate a list of all macro definitions. The preprocessor output is discarded.

With **--preprocess=+make** the compiler will generate dependency lines that can be used in a Makefile. The information is written to a file with extension `.d`. The preprocessor output is discarded. The default target name is the basename of the input file, with the extension `.obj`. With the option **--make-target** you can specify a target name which overrules the default target name.

With **--preprocess=+noline** you tell the preprocessor to strip the #line source position information (lines starting with `#line`). These lines are normally processed by the assembler and not needed in the preprocessed output. When you leave these lines out, the output is easier to read.

Example

```
cc166 --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

Menu entry

1. From the **Window** menu, select **Preferences**.

    The Preferences dialog appears.

2. Select **TASKING C166 Preferences**.

3. Click the **Add** button to add additional processor definition files.

Command line syntax

--processors=*file*

Description

With this option you can specify an additional XML file with processor definitions.

The standard list of supported processors is defined in the file `processors.xml`. This file defines for each processor its full name (for example, XC2287-72F), the base CPU name (for example, xc2287), the core settings (for example, xc16x), the on-chip flash settings, the list of silicon bugs for that processor. Each processor also defines an option to supply to the linker for preprocessing the LSL file for the applicable on-chip memory definitions.

The control program reads the specified *file* after the `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:

```
cc166 --processors=new-processors.xml --cpu=NEW-PROC test.c
```

Related information

Control program option **--cpu** (Select processor)

Control program option **--core** (Select the core)

Control program option: --profile (-p)

Menu entry

1. Select **C/C++ Compiler » Debugging**.

2. Enable or disable **Static profiling**.

3. Enable or disable one or more of the following **Generate profiling information** options (dynamic profiling):

    *   **for block counters** (not in combination with Call graph or Function timers)

    *   **to build a call graph** (not in combination with Block counters)

    *   **for function counters**

    *   **for function timers** (not in combination with Block counters/Function counters)

> Note that the more detailed information you request, the larger the overhead in terms of execution time, code size and heap space needed. The option **--debug** does not affect profiling, execution time or code size.

Command line syntax

--profile[**=**$flag$,...]

-p[$flags$]

Use the following option for a predefined set of flags:

--profile=g	**-pg**	Profiling with call graph and function timers.
		Alias for: **-pBcFSt**

You can set the following flags:

+/-block	**b/B**	block counters
+/-callgraph	**c/C**	call graph
+/-function	**f/F**	function counters
+/-static	**s/S**	static profile generation
+/-time	**t/T**	function timers

Default (without flags): **-pBCfST**

Description

Profiling is the process of collecting statistical data about a running application. With these data you can analyze which functions are called, how often they are called and what their execution time is.

Several methods of profiling exist. One method is *code instrumentation* which adds code to your application that takes care of the profiling process when the application is executed. Another method is *static profiling*.

For an extensive description of profiling refer to Chapter 6, *Profiling*.

You can obtain the following profiling data (see flags above):

Block counters (not in combination with Call graph or Function timers)

This will instrument the code to perform basic block counting. As the program runs, it counts the number of executions of each branch in an if statement, each iteration of a for loop, and so on. Note that though you can combine Block counters with Function counters, this has no effect because Function counters is only a subset of Block counters.

Call graph (not in combination with Block counters)

This will instrument the code to reconstruct the run-time call graph. As the program runs it associates the caller with the gathered profiling data.

Function counters

This will instrument the code to perform function call counting. This is a subset of the basic Block counters.

Function timers (not in combination with Block counters/Function counters)

This will instrument the code to measure the time spent in a function. This includes the time spent in all sub functions (callees).

Static profiling

With this option you do not need to run the application to get profiling results. The compiler generates profiling information at compile time, without adding extra code to your application.

> Note that the more detailed information you request, the larger the overhead in terms of execution time, code size and heap space needed. The option **Generate symbolic debug information** (**--debug**) does not affect profiling, execution time or code size.

The control program automatically specifies the corresponding profiling libraries to the linker.

Example

To generate block count information for the module `test.c` during execution, compile as follows:

```
cc166 --profile=+block test.c
```

In this case the control program tells the linker to link the library `c166pbn.lib`.

Related information

Chapter 6, *Profiling*

Control program option: --show-c++-warnings

Menu entry

-

Command line syntax

`--show-c++-warnings`

Description

The C++ compiler may generate a compiled C++ file (`.ic`) that causes warnings during compilation or assembling. With this option you tell the control program to show these warnings. By default C++ warnings are suppressed.

Related information

-

Control program option: --signed-bitfields

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Treat "int" bit-fields as signed**.

Command line syntax

```
--signed-bitfields
```

Description

For bit-fields it depends on the implementation whether a plain `int` is treated as `signed int` or `unsigned int`. By default an `int` bit-field is treated as `unsigned int`. This offers the best performance. With this option you tell the compiler to treat `int` bit-fields as `signed int`. In this case, you can still add the keyword `unsigned` to treat a particular `int` bit-field as `unsigned`.

Related information

C compiler option **--signed-bitfields**

Section 1.1, *Data Types*

Control program option: --uchar (-u)

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Treat "char" variables as unsigned**.

Command line syntax

`--uchar`

`-u`

Description

By default `char` is the same as specifying `signed char`. With this option `char` is the same as `unsigned char`. This option is passed to both the C++ compiler and the C compiler.

Related information

Section 1.1, *Data Types*

Control program option: --undefine (-U)

Menu entry

1. Select **C/C++ Compiler » Preprocessing**

    The Defined symbols box shows the symbols that are currently defined.

2. To remove a defined symbol, select the symbol in the **Defined symbols** box and click on the **Delete** button.

Command line syntax

--undefine=_macro_name_

-U_macro_name_

Description

With this option you can undefine an earlier defined macro as with `#undef`. This option is for example useful to undefine predefined macros.

The following predefined ISO C standard macros cannot be undefined:

`__FILE__`	current source filename
`__LINE__`	current source line number (int type)
`__TIME__`	hh:mm:ss
`__DATE__`	Mmm dd yyyy
`__STDC__`	level of ANSI standard

The control program passes the option **--undefine** (**-U**) to the compiler.

Example

To undefine the predefined macro `__TASKING__`:

```
cc166 --undefine=__TASKING__ test.c
```

Related information

Control program option **--define** (Define preprocessor macro)

Section 1.7, *Predefined Preprocessor Macros*

Control program option: --user-stack

Menu entry

1. Select **C/C++ Compiler » Memory Model**.

2. Enable the option **Use user stack for return addresses**.

Command line syntax

`--user-stack`

Description

With this option the function return address is stored on the user stack instead of on the system stack. Also the user stack run-time libraries are used and the user stack calling convention is used to call the run-time library functions.

The control program also tells the linker to link the appropriate user-stack variant of the libraries.

Related information

Section 1.12, *Functions*

Control program option: --verbose (-v)

Menu entry

1. Select **Global Options**.

2. Enable the option **Verbose mode of control program**.

Command line syntax

`--verbose`

`-v`

Description

With this option you put the control program in verbose mode. The control program performs 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)

Menu entry

-

Command line syntax

`--version`

`-V`

Description

Display version information. The control program ignores all other options or input files.

Related information

-

Control program option: --warnings-as-errors

Menu entry

1. Select **Global Options**.

2. Enable the option **Treat warnings as errors**.

Command line syntax

```
--warnings-as-errors[=number[-number],...]
```

Description

If one of the tools encounters an error, it stops processing the file(s). With this option you tell the tools to treat warnings as errors or treat specific control program warning messages as errors:

• If you specify this option but without numbers, all warnings are treated as errors.

• If you specify this option with a number or a range, only the specified control program warnings are treated as an error. You can specify the option **--warnings-as-errors=**number multiple times.

Use one of the **--pass-**tool options to pass this option directly to a tool when a specific warning for that tool must be treated as an error. For example, use **--pass-c=--warnings-as-errors=**number to treat a specific C compiler warning as an error.

Related information

Control program option **--no-warnings** (Suppress some or all warnings)

Control program option **--pass** (Pass option to tool)

12.7. Make Utility Options

When you build a project in Eclipse, Eclipse generates a makefile and uses the make utility **mk166** to build all your files. However, you can also use the make utility directly from the command line to build your project.

The invocation syntax is:

mk166 [*option*...] [*target*...] [*macro*=*def*]

This section describes all options for the make utility. The make utility is a command line tool so there are no equivalent options in Eclipse.

For detailed information about the make utility and using makefiles see Section 9.2, *Make Utility mk166*.

Defining Macros

Command line syntax

macro_name[**=***macro_definition*]

Description

With this argument you can define a macro and specify it to the make utility.

A macro definition remains in existence during the execution of the makefile, even when the makefile recursively calls the make utility again. In the recursive call, the macro acts as an environment variable. This means that it is overruled by definitions in the recursive call. Use the option **-e** to prevent this.

You can specify as many macros as you like. If the command line exceeds the limit of the operating system, you can define the macros in an option file which you then must specify to the make utility with the option **-m**) *file*.

Defining macros on the command line is, for example, useful in combination with conditional processing as shown in the example below.

Example

Consider the following makefile with conditional rules to build a demo program and a real program:

```
ifdef DEMO        # the value of DEMO is of no importance
   real.elf : demo.obj main.obj
             lk166 demo.obj main.obj -lcn -lfpn -lrtn
else
   real.elf : real.obj main.obj
             lk166 real.obj main.obj -lcn -lfpn -lrtn
endif
```

You can now use a macro definition to set the DEMO flag:

```
mk166 real.elf DEMO=1
```

In both cases the absolute object file `real.elf` is created but depending on the DEMO flag it is linked with `demo.obj` or with `real.obj`.

Related information

Make utility option **-e** (Environment variables override macro definitions)

Make utility option **-m** (Name of invocation file)

Make utility option: -?

Command line syntax

`-?`

Description

Displays an overview of all command line options.

Example

The following invocation displays a list of the available command line options:

`mk166 -?`

Related information

-

Make utility option: -a

Command line syntax

`-a`

Description

Normally the make utility rebuilds only those files that are out of date. With this option you tell the make utility to rebuild *all* files, without checking whether they are out of date.

Example

```
mk166 -a
```

Rebuilds all your files, regardless of whether they are out of date or not.

Related information

-

Make utility option: -c

Command line syntax

`-c`

Description

Eclipse uses this option when you create sub-projects. In this case the make utility calls another instance of the make utility for the sub-project. With the option **-c**, the make utility runs as a child process of the current make.

The option **-c** overrules the option **-err**.

Example

```
mk166 -c
```

The make utility runs its commands as a child processes.

Related information

Make utility option **-err** (Redirect error message to file)

Make utility option: -D / -DD

Command line syntax

`-D`
`-DD`

Description

With the option **-D** the make utility prints every line of the makefile to standard output as it is read by **mk166**.

With the option **-DD** not only the lines of the makefile are printed but also the lines of the `mk166.mk` file (implicit rules).

Example

`mk166 -D`

Each line of the makefile that is read by the make utility is printed to standard output (usually your screen).

Related information

-

Make utility option: -d/ -dd

Command line syntax

```
-d
-dd
```

Description

With the option **-d** the make utility shows which files are out of date and thus need to be rebuild. The option **-dd** gives more detail than the option **-d**.

Example

```
mk166 -d
```

Shows which files are out of date and rebuilds them.

Related information

-

Make utility option: -e

Command line syntax

-e

Description

If you use macro definitions, they may overrule the settings of the environment variables. With the option **-e**, the settings of the environment variables are used even if macros define otherwise.

Example

```
mk166 -e
```

The make utility uses the settings of the environment variables regardless of macro definitions.

Related information

-

Make utility option: -err

Command line syntax

-err *file*

Description

With this option the make utility redirects error messages and verbose messages to a specified file.

With the option **-s** the make utility only displays error messages.

Example

mk166 -err error.txt

The make utility writes messages to the file error.txt.

Related information

Make utility option **-s** (Do not print commands before execution)

Make utility option **-c** (Run as child process)

Make utility option: -f

Command line syntax

-f *my_makefile*

Description

By default the make utility uses the file `makefile` to build your files.

With this option you tell the make utility to use the specified file instead of the file `makefile`. Multiple **-f** options act as if all the makefiles were concatenated in a left-to-right order.

If you use '-' instead of a makefile name it means that the information is read from `stdin`.

Example

```
mk166 -f mymake
```

The make utility uses the file `mymake` to build your files.

Related information

-

Make utility option: -G

Command line syntax

-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.

Example

Suppose your makefile and other files are stored in the directory `..\myfiles`. You can call the make utility, for example, as follows:

```
mk166 -G ..\myfiles
```

Related information

-

Make utility option: -i

Command line syntax

`-i`

Description

When an error occurs during the make process, the make utility exits with a certain exit code.

With the option **-i**, the make utility exits without an error code, even when errors occurred.

Example

```
mk166 -i
```

The make utility exits without an error code, even when an error occurs.

Related information

-

Make utility option: -K

Command line syntax

`-K`

Description

With this option the make utility keeps temporary files it creates during the make process. The make utility stores temporary files in the directory that you have specified with the environment variable TMPDIR or in the default 'temp' directory of your system when the TMPDIR environment variable is not specified.

Example

```
mk166 -K
```

The make utility preserves all temporary files.

Related information

-

Make utility option: -k

Command line syntax

`-k`

Description

When during the make process the make utility encounters an error, it stops rebuilding your files.

With the option **-k**, the make utility only stops building the target that produced the error. All other targets defined in the makefile are built.

Example

```
mk166 -k
```

If the make utility encounters an error, it stops building the current target but proceeds with the other targets that are defined in the makefile.

Related information

Make utility option **-S** (Undo the effect of **-k**)

Make utility option: -m

Command line syntax

–m *file*

Description

Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the make utility.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option **-m** multiple times.

If you use '-' instead of a filename it means that the options are read from stdin.

Format of an option file

• Multiple arguments on one line in the option file are allowed.

• To include whitespace in an argument, surround the argument with single or double quotes.

• If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

```
"This has a single quote ' embedded"

'This has a double quote " embedded'

'This has a double quote " and a single quote '"' embedded"
```

Note that adjacent strings are concatenated.

• When a text line reaches its length limit, use a **** to continue the line. Whitespace between quotes is preserved.

```
"This is a continuation \
line"

        -> "This is a continuation line"
```

• It is possible to nest command line files up to 25 levels.

Example

Suppose the file myoptions contains the following lines:

```
-k
-err errors.txt
test.elf
```

Specify the option file to the make utility:

```
mk166 -m myoptions
```

This is equivalent to the following command line:

```
mk166 -k -err errors.txt test.elf
```

Related information

-

Make utility option: -n

Command line syntax

`-n`

Description

With this option you tell the make utility to perform a dry run. The make utility shows what it would do but does not actually perform these tasks.

This option is for example useful to quickly inspect what would happen if you call the make utility.

Example

`mk166 -n`

The make utility does not perform any tasks but displays what it would do if called without the option **-n**.

Related information

Make utility option **-s** (Do not print commands before execution)

Make utility option: -p

Command line syntax

`-p`

Description

Normally, if a command in a target rule in a makefile returns an error or when the target construction is interrupted, the make utility removes that target file. With this option you tell the make utility to make all target files precious. This means that all dependency files are never removed.

Example

```
mk166 -p
```

The make utility never removes target dependency files.

Related information

Special target `.PRECIOUS` in Section 9.2.2.1, *Targets and Dependencies*

Make utility option: -q

Command line syntax

`-q`

Description

With this option the make utility does not perform any tasks but only returns an exit code. A zero status indicates that all target files are up to date, a non-zero status indicates that some or all target files are out of date.

Example

```
mk166 -q
```

The make utility only returns an error code that indicates whether all target files are up to date or not. It does not rebuild any files.

Related information

-

Make utility option: -r

Command line syntax

`-r`

Description

When you call the make utility, it first reads the implicit rules from the file `mk166.mk`, then it reads the makefile with the rules to build your files. (The file `mk166.mk` is located in the `\etc` directory of the toolset.)

With this option you tell the make utility not to read `mk166.mk` and to rely fully on the make rules in the makefile.

Example

`mk166 -r`

The make utility does not read the implicit make rules in `mk166.mk`.

Related information

-

Make utility option: -S

Command line syntax

`-S`

Description

With this option you cancel the effect of the option **-k**. This is only necessary in a recursive make where the option **-k** might be inherited from the top-level make via MAKEFLAGS or if you set the option **-k** in the environment variable MAKEFLAGS.

With this option you tell the make utility not to read `mk166.mk` and to rely fully on the make rules in the makefile.

Example

`mk166 -S`

The effect of the option **-k** is cancelled so the make utility stops with the make process after it encounters an error.

The option **-k** in this example may have been set with the environment variable MAKEFLAGS or in a recursive call to **mk166** in the makefile.

Related information

Make utility option **-k** (On error, abandon the work for the current target only)

Make utility option: -s

Command line syntax

`-s`

Description

With this option you tell the make utility to perform its tasks without printing the commands it executes. Error messages are normally printed.

Example

```
mk166 -s
```

The make utility rebuilds your files but does not print the commands it executes during the make process.

Related information

Make utility option **-n** (Perform a dry run)

Make utility option: -t

Command line syntax

`-t`

Description

With this option you tell the make utility to *touch* the target files, bringing them up to date, rather than performing the rules to rebuild them.

Example

`mk166 -t`

The make utility updates out-of-date files by giving them a new date and time stamp. The files are not actually rebuild.

Related information

-

Make utility option: -time

Command line syntax

`-time`

Description

With this option you tell the make utility to display the current date and time on standard output.

Example

```
mk166 -time
```

The make utility displays the current date and time and updates out-of-date files.

Related information

-

Make utility option: -V

Command line syntax

`-V`

Description

Display version information. The make utility ignores all other options or input files.

Example

`mk166 -V`

The make utility displays the version information but does not perform any tasks.

```
TASKING VX-toolset for C166: program builder    vx.yrz Build nnn
Copyright 2004-year Altium BV                    Serial# 00000000
```

Related information

-

Make utility option: -W

Command line syntax

-W *target*

Description

With this option the make utility considers the specified target file always as up to date and will not rebuild it.

Example

```
mk166 -W test.elf
```

The make utility rebuilds out of date targets in the makefile except the file test.elf which is considered now as up to date.

Related information

-

Make utility option: -w

Command line syntax

`-w`

Description

With this option the make utility sends error messages and verbose messages to standard output. Without this option, the make utility sends these messages to standard error.

This option is only useful on UNIX systems.

Example

```
mk166 -w
```

The make utility sends messages to standard out instead of standard error.

Related information

-

Make utility option: -x

Command line syntax

`-x`

Description

With this option the make utility shows extended error messages. Extended error messages give more detailed information about the exit status of the make utility after errors.

Example

```
mk166 -x
```

If errors occur, the make utility gives extended information.

Related information

-

12.8. Parallel Make Utility Options

When you build a project in Eclipse, Eclipse generates a makefile and uses the make utility **amk** to build all your files. However, you can also use the make utility directly from the command line to build your project.

The invocation syntax is:

amk [*option*...] [*target*...] [*macro=def*]

This section describes all options for the parallel make utility.

For detailed information about the parallel make utility and using makefiles see Section 9.3, *Make Utility amk*.

Parallel make utility option: -?

Command line syntax

-?

Description

Displays an overview of all command line options.

Example

The following invocation displays a list of the available command line options:

```
amk -?
```

Related information

-

Parallel make utility option: -a

Command line syntax

`-a`

Description

Normally the make utility rebuilds only those files that are out of date. With this option you tell the make utility to rebuild *all* files, without checking whether they are out of date.

Example

```
amk -a
```

Rebuilds all your files, regardless of whether they are out of date or not.

Related information

-

Parallel make utility option: -f

Command line syntax

-f *my_makefile*

Description

By default the make utility uses the file `makefile` to build your files.

With this option you tell the make utility to use the specified file instead of the file `makefile`. Multiple **-f** options act as if all the makefiles were concatenated in a left-to-right order.

If you use '-' instead of a makefile name it means that the information is read from `stdin`.

Example

```
amk -f mymake
```

The make utility uses the file `mymake` to build your files.

Related information

-

Parallel make utility option: -G

Command line syntax

-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: -j / -J

Menu

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

   The Properties dialog appears.

2. In the left pane, select **C/C++ Build**.

   In the right pane the C/C++ Build page appears.

3. On the Behaviour tab, select **Use parallel build**.

4. You can specify the number of parallel jobs, or you can use an optimal number of jobs. In the last case, **amk** will fork as many jobs in parallel as cores are available.

Command line syntax

-j[*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: -k

Command line syntax

`-k`

Description

When during the make process the make utility encounters an error, it stops rebuilding your files.

With the option **-k**, the make utility only stops building the target that produced the error. All other targets defined in the makefile are built.

Example

```
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: -n

Command line syntax

`-n`

Description

With this option you tell the make utility to perform a dry run. The make utility shows what it would do but does not actually perform these tasks.

This option is for example useful to quickly inspect what would happen if you call the make utility.

Example

`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: -s

Command line syntax

`-s`

Description

With this option you tell the make utility to perform its tasks without printing the commands it executes. Error messages are normally printed.

Example

```
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: -V

Command line syntax

`-v`

Description

Display version information. The make utility ignores all other options or input files.

Related information

-

12.9. Archiver Options

The archiver and library maintainer **ar166** 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:

ar166 *key_option* [*sub_option...*] *library* [*object_file*]

This section describes all options for the archiver. Some suboptions can only be used in combination with certain key options. They are described together. Suboptions that can always be used are described separately.

For detailed information about the archiver, see Section 9.4, *Archiver*.

Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (**-**) character, long option names always begin with two minus (**--**) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

Overview of the options of the archiver utility

The following archiver options are available:

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

ar166 --delete mylib.lib obj1.obj obj2.obj

The archiver deletes obj1.obj and obj2.obj from the library mylib.lib.

ar166 -d -v mylib.lib obj1.obj obj2.obj

The archiver deletes obj1.obj and obj2.obj from the library mylib.lib and displays which files are removed.

Related information

-

Archiver option: --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

```
ar166 --dump mylib.lib obj1.obj > file.obj
```

The archiver prints the file `obj1.obj` to standard output where it is redirected to the file `file.obj`. The effect of this example is very similar to extracting a file from the library but in this case the 'extracted' file gets another name.

Related information

-

Archiver option: --extract (-x)

Command line syntax

--extract [**--modtime**] [**--verbose**]

-x [**-o**] [**-v**]

Description

Extract an existing module from the library.

--modtime	**-o**	Give the extracted object module the same date as the last-modified date that was recorded in the library. Without this suboption it receives the last-modified date of the moment it is extracted.
--verbose	**-v**	Verbose: the archiver shows which files are extracted.

Example

To extract the file obj1.obj from the library mylib.lib:

ar166 --extract mylib.lib obj1.obj

If you do not specify an object module, all object modules are extracted:

ar166 –x mylib.lib

Related information

-

Archiver option: --help (-?)

Command line syntax

--help[**=**_item_]

-?

You can specify the following argument:

options   Show extended option descriptions

Description

Displays an overview of all command line options. When you specify the argument **options** you can list detailed option descriptions.

Example

The following invocations all display a list of the available command line options:

```
ar166 -?
ar166 --help
ar166
```

To see a detailed description of the available options, enter:

```
ar166 --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.lib` contains the following objects (see option **--print**):

```
obj1.obj
obj2.obj
obj3.obj
```

To move `obj1.obj` to the end of `mylib.lib`:

```
ar166 --move mylib.lib obj1.obj
```

To move `obj3.obj` just before `obj2.obj`:

```
ar166 -m -b obj3.obj mylib.lib obj2.obj
```

The library `mylib.lib` after these two invocations now looks like:

```
obj3.obj
obj2.obj
obj1.obj
```

Related information

Archiver option **--print** (**-t**) (Print library contents)

Archiver option: --option-file (-f)

Command line syntax

--option-file=$file$

-f $file$

Description

Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the archiver.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option **--option-file** (**-f**) multiple times.

If you use '-' instead of a filename it means that the options are read from stdin.

Format of an option file

- Multiple arguments on one line in the option file are allowed.

- To include whitespace in an argument, surround the argument with single or double quotes.

- If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

  ```
  "This has a single quote ' embedded"

  'This has a double quote " embedded'

  'This has a double quote " and a single quote '"' embedded"
  ```

- When a text line reaches its length limit, use a **** to continue the line. Whitespace between quotes is preserved.

  ```
  "This is a continuation \
  line"

          -> "This is a continuation line"
  ```

- It is possible to nest command line files up to 25 levels.

Example

Suppose the file myoptions contains the following lines:

```
-x mylib.lib obj1.obj
-w5
```

Specify the option file to the archiver:

```
ar166 --option-file=myoptions
```

This is equivalent to the following command line:

```
ar166 -x mylib.lib obj1.obj -w5
```

Related information

-

Archiver option: --print (-t)

Command line syntax

--print [**--symbols=0**│**1**]

-t [**-s0**│**-s1**]

Description

Print a table of contents of the library to standard output. With the suboption **-s0** the archiver displays all symbols per object file.

--symbols=0	**-s0**	Displays per object the name of the object itself and all symbols in the object.
--symbols=1	**-s1**	Displays the symbols of all object files in the library in the form *library_name*:*object_name*:*symbol_name*

Example

```
ar166 --print mylib.lib
```

The archiver prints a list of all object modules in the library `mylib.lib`:

```
ar166 -t -s0 mylib.lib
```

The archiver prints per object all symbols in the library. For example:

```
cstart.obj
   symbols:
        __cstart
        .vector.0
        _cstart_trap
```

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.lib` contains the following object (see option **--print**):

`obj1.obj`

To add `obj2.obj` to the end of `mylib.lib`:

`ar166 --replace mylib.lib obj2.obj`

To insert `obj3.obj` just before `obj2.obj`:

`ar166 -r -b obj2.obj mylib.lib obj3.obj`

The library `mylib.lib` after these two invocations now looks like:

```
obj1.obj
obj3.obj
obj2.obj
```

Creating a new library

To *create a new library file*, add an object file and specify a library that does not yet exist:

```
ar166 --replace obj1.obj newlib.lib
```

The archiver creates the library `newlib.lib` and adds the object `obj1.obj` to it.

To *create a new library file and overwrite an existing library*, add an object file and specify an existing library with the suboption **--create** (**-c**):

```
ar166 -r -c obj1.obj mylib.lib
```

The archiver overwrites the library `mylib.lib` and adds the object `obj1.obj` to it. The new library `mylib.lib` only contains `obj1.obj`.

Related information

Archiver option **--print** (**-t**) (Print library contents)

Archiver option: --version (-V)

Command line syntax

`--version`

`-V`

Description

Display version information. The archiver ignores all other options or input files.

Example

`ar166 -V`

The archiver displays the version information but does not perform any tasks.

```
TASKING VX-toolset for C166: ELF archiver    vx.yrz Build nnn
Copyright 2004-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:

```
ar166 --extract --warning=5 mylib.lib obj1.obj
```

Related information

-

12.10. HLL Object Dumper Options

The high level language (HLL) dumper **hldump166** is a program to dump information about an object file.

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:

```
hldump166 -FdhMsy test.elf
hldump166 --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 9.5.2, _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`, `.dw`, `.dl` 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

```
hldump166 -F2 --section=near__1_str hello.elf

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

    .section data, 'near__1_str'
    .org 00000200
    .db 48,65,6C,6C,6F,20,77,6F,72,6C,64,21,0A,00      ; Hello world!..
    .endsec

hldump166 -F2 --section=near__1_str --data-dump-format=hex hello.elf

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

                        section 5 (near__1_str):
00000200 48 65 6C 6C 6F 20 77 6F 72 6C 64 21 0A 00      Hello world!..
```

Related information

Section 9.5.2, *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

`hldump166 --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 9.5.2, *HLL Dump Output Format*

HLL object dumper option: --expand-symbols (-e)

Command line syntax

--expand-symbols[**=**_flag_]

-e[_flag_]

You can specify one of the following flags:

+/-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

```
hldump166 -F1 hello.elf

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

0000F738     10 struct          __near _dbg_request [dbg.c]

hldump166 -e -F1 hello.elf

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

0000F738     10 struct          __near _dbg_request [dbg.c]
0000F738      2   int                  _errno
0000F73A      1   enum                 nr
0000F73C      6   union                u
0000F73C      2     struct               exit
0000F73C      2       int                  status
0000F73C      4     struct               open
0000F73C      2       const char       * pathname
0000F73E      2       unsigned short int   flags
   ...

hldump166 -ef -F1 hello.elf

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

0000F738     24 struct          __near _dbg_request [dbg.c]
0000F738      4   int                  _dbg_request._errno
0000F73A      4   enum                 _dbg_request.nr
0000F73C     16   union                _dbg_request.u
0000F73C      4     struct               _dbg_request.u.exit
```

```
0000F73C      4      int                  _dbg_request.u.exit.status
0000F73C      8    struct                 _dbg_request.u.open
0000F73C      4      const char        * _dbg_request.u.open.pathname
0000F73E      2      unsigned short int   _dbg_request.u.open.flags
   ...
```

Related information

Section 9.5.2, *HLL Dump Output Format*

HLL object dumper option: --help (-?)

Command line syntax

`--help`

`-?`

Description

Displays an overview of all command line options.

Example

The following invocations all display a list of the available command line options:

```
hldump166 -?
hldump166 --help
hldump166
```

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:

```
hldump166 --option-file=myoptions
```

This is equivalent to the following command line:

```
hldump166 --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

hldump166 --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: --print-mangled-symbols (-P)

Command line syntax

```
--print-mangled-symbols
```

```
-P
```

Description

The C++ compiler can generate unreadable symbol names. These symbols cannot easily be related to your C++ source file anymore. Therefore the HLL dumper by default demangles C++ function names and variable names in the HLL symbol table. With this option you can override this default setting and print the mangled names instead.

Example

```
hldump166 hellocpp.elf

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

Address     Size HLL Type           Name
00400CD8       8 void                __register_finalization_routine()

hldump166 --print-mangled-symbols hellocpp.elf

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

Address     Size HLL Type           Name
00400CD8       8 void                _Z31__register_finalization_routinev()
```

Related information

-

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 9.5.2, *HLL Dump Output Format*

HLL object dumper option: --source-lookup-path (-L)

Command line syntax

--source-lookup-path=$path$

-L$path$

Description

With this option you can specify an additional path where your source files are located. If you want to specify multiple paths, use the option **--source-lookup-path** for each separate path.

The order in which the HLL object dumper will search for source files when intermixed disassembly is used, is:

1. The path obtained from the HLL debug information.

2. The path that is specified with the option **--source-lookup-path**. If multiple paths are specified, the paths will be searched for in the order in which they are given on the command line.

Example

Suppose you call the HLL object dumper as follows:

```
hldump166 --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

```
hldump166 -F2 hello.elf

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

                                  .section code, '.vector.0'
00000000 030040FA                 jmps    __cstart
                                  ; End of section

hldump166 --symbols=none -F2 hello.elf

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

                                  .section code, '.vector.0'
00000000 030040FA                 jmps    0x400300
                                  ; End of section
```

Related information

Section 9.5.2, *HLL Dump Output Format*

HLL object dumper option: --version (-V)

Command line syntax

```
--version
```

```
-v
```

Description

Display version information. The HLL object dumper ignores all other options or input files.

Related information

-

HLL object dumper option: --xml-base-filename (-X)

Command line syntax

`--xml-base-filename`

`-X`

Description

With this option the `<File name>` field in the XML output only contains the filename of the object file. By default, any path name, if present, is printed as well.

Example

`hldump166 --output-type=xml --output=hello.xml ../hello.elf`

The field `<File name="../hello.elf">` is used in `hello.xml`.

`hldump166 --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 13. 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 13.1, *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 13.2, *C Library Reentrancy*, gives an overview of which functions are reentrant and which are not.

The following libraries are included in the C166 toolset. Both Eclipse and the control program **cc166** automatically select the appropriate libraries depending on the specified options.

C library

Libraries	Description
c166c*m*[n][u][s].lib	C libraries for each model *m*: n (near), f (far), s (shuge), h (huge) Optional letters: n = near functions u = user stack s = single precision floating-point
c166fp*m*[n][u][t].lib	Floating-point libraries for each model *m*: n, f, s, h Optional letters: n = near functions u = user stack t = trapping
c166rt*m*[n][u].lib	Run-time libraries for each model *m*: n, f, s, h Optional letters: n = near functions u = user stack
c166pb*m*[n][u].lib c166pc*m*[n][u].lib c166pct*m*[n][u].lib c166pd*m*[n][u].lib c166pt*m*[n][u].lib	Profiling libraries for each model *m*: n, f, s, h pb = block/function counter pc = call graph pct = call graph and timing pd = dummy pt = function timing Optional letters: n = near functions u = user stack

C++ Library

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

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

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

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

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

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

Libraries	Description
c166cp*m*[u][x].lib	C++ libraries for each model *m*: n, f, s, h Optional letters: u = user stack x = exception handling
c166stl*m*x.lib	STLport C++ libraries for each model *m*: n, f, s, h Optional letters: x = exception handling

Sources for the STLport C++ libraries are delivered in the `lib\src.stl` directory with makefiles to build the libraries.

To build an STLport library

1. Change to the directory *installdir*`\lib\src.stl\[p][1][2]\c166stl?x`, depending on the library set used by your project.

2. Run the makefile by executing *installdir*`\bin\mk166.exe` without arguments.

3. Copy the generated C++ library `c166stl?x.lib` to the corresponding directory *installdir*`\lib\[p][1][2]`.

where,

?	represents the memory model (n=near, f=far, h=huge, s=segmented huge)
[1]	libraries for C16x/ST10/ST10mac architectures
[2]	libraries for extended XC16x/Super10 architectures
[p]	protected libraries for CPU functional problems
x	STLport library with exception handling

13.1. 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.

13.1.1. assert.h

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

13.1.2. complex.h

The complex number *z* is also written as *x+yi* where *x* (the real part) and *y* (the imaginary part) are real numbers of types `float`, `double` or `long double`. The real and imaginary part can be stored in structs or in arrays. This implementation uses arrays because structs may have different alignments.

The header file `complex.h` also defines the following macros for backward compatibility:

```
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)

13.1.3. cstart.h

The header file cstart.h controls the system startup code's general settings and register initializations. It contains defines only, no functions.

13.1.4. 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.

13.1.5. 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

13.1.6. 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 |

13.1.7. fcntl.h

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

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

13.1.8. fenv.h

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

| fegetenv | Stores the current floating-point environment. (*Not implemented*) |

`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.

13.1.9. float.h

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

> `float.h` used to contain prototypes for the functions `copysign(f)`, `isinf(f)`, `isfinite(f)`, `isnan(f)` and `scalb(f)`. These functions have accordingly to the ISO C99 standard been moved to the header file `math.h`. See also Section 13.1.17, *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 `scalb` in Section 13.1.17, *math.h and tgmath.h*)

13.1.10. fpbits.h

The header file `fpbits.h` contains definitions for the internals of the run-time floating-point library implementation. This header file is not defined in ISO C99.

13.1.11. 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`)

13.1.12. 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)*

13.1.13. 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   ^=
```

13.1.14. limits.h

Contains the sizes of integral types, defined as macros.

13.1.15. 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.

13.1.16. 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 13.1.25, *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.

Besides these functions, the C166 library supports __near, __far, __huge and __shuge variants of the memory allocation functions.

`__nmalloc`
`__ncalloc`
`__nfree`
`__nrealloc`
	Allocation in near memory. These functions are default for the near memory model.

`__fmalloc`
`__fcalloc`
`__ffree`
`__frealloc`
	Allocation in far memory. These functions are default for the far memory model.

`__smalloc`
`__scalloc`
`__sfree`
`__srealloc`
	Allocation in segmented huge memory. These functions are default for the shuge memory model.

`__hmalloc`
`__hcalloc`
`__hfree`
`__hrealloc`
	Allocation in huge memory, *size* is limited to 64 kB. These functions are default for the huge memory model.

`__xhmalloc`	Extended allocation in huge memory, *size* is limited to 32-bit.
`__xhcalloc`	
`__xhfree`	
`__xhrealloc`	

Based on the memory model the compiler chooses the correct version. For example, if you use `malloc()` in your source and you compile for the near memory model, the compiler uses the function `__nmalloc()`.

13.1.17. 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`.
`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 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.

797

math.h			tgmath.h	Description
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 $f \cdot 2^n = x$. Returns *f*, stores *n*.
ldexp	ldexpf	ldexpl	ldexp	Inverse of `frexp`. Returns the result of $x \cdot 2^n$. (`x` and `n` are both arguments).
modf	modff	modfl	–	Splits a float `x` into fraction *f* and integer *n*, so that: $|f| < 1.0$ and $f+n=x$. Returns *f*, stores *n*.
scalbn	scalbnf	scalbnl	scalbn	Computes the result of $x \cdot FLT\_RADIX^n$. 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*)

math.h			tgmath.h	Description
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 ($\lvert x \rvert$). (`abs`, `labs`, `llabs`, `div`, `ldiv`, `lldiv` are defined in `stdlib.h`)
fma	fmaf	fmal	fma	Floating-point multiply add. Returns `x*y+z`. (*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	copysignll	copysign	Returns the value of `x` with the sign of `y`.
nan	nanf	nanl	–	Returns a quiet NaN, if available, with content indicated through *tagp*. (*Not implemented*)

math.h			tgmath.h	Description
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*)
erfc	erfcf	erfcl	erc	Computes the complementary error function of x. (*Not implemented*)
lgamma	lgammaf	lgammal	lgamma	Computes the $*\log_e|\Gamma(x)|$ (*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)

math.h	tgmath.h	Description
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.

13.1.18. 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()`.

13.1.19. 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.

13.1.20. 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. It's 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.

13.1.21. 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
```

13.1.22. 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.

13.1.23. stdint.h

See Section 13.1.11, *inttypes.h and stdint.h*

13.1.24. 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.

stdio.h	Description
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)`.

stdio.h	Description
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
%	No argument is converted, a '%' is printed.

printf conversion characters

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

The format string can contain :

- Blanks or tabs, which are skipped.

- Normal characters (not '%'), which should be matched exactly in the input stream.

- Conversion specifications, starting with a '%' character.

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

- A '*', meaning that no assignment is done for this field.

- A number specifying the maximum field width.

- The conversion characters `d`, `i`, `n`, `o`, `u` and `x` may be preceded by 'h' if the argument is a pointer to `short` rather than `int`, or by 'hh' if the argument is a pointer to `char`, or by 'l' (letter ell) if the argument is a pointer to `long` or by 'll' for a pointer to `long long`, 'j' for a pointer to `intmax_t` or `uintmax_t`, 'z' for a pointer to `size_t` or 't' for a pointer to `ptrdiff_t`. The conversion characters `e`, `f`, and `g` may be preceded by 'l' if the argument is a pointer to `double` rather than `float`, and by 'L' for a pointer to a `long double`.

- A conversion specifier. '*', maximum field width and length modifier are optional, the conversion character is not. The conversion character must be one of the following, if a character following '%' is not in the list, the behavior is undefined.

Length specifier and length modifier are optional, the conversion character is not. The conversion character must be one of the following, if a character following '%' is not in the list, the behavior is undefined.

Character	Scanned as
d	int, signed decimal.
i	int, the integer may be given octal (i.e. a leading 0 is entered) or hexadecimal (leading "0x" or "0X"), or just decimal.
o	int, unsigned octal.
u	int, unsigned decimal.
x	int, unsigned hexadecimal in lowercase or uppercase.
c	single character (converted to unsigned char).
s	char *, a string of non white space characters. The argument should point to an array of characters, large enough to hold the string and a terminating NULL character.
f, F	float
e, E	float
g, G	float
a, A	float
n	int *, the number of characters written so far is written into the argument. No scanning is done.
p	pointer; hexadecimal value which must be entered without 0x- prefix.
[...]	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.

stdio.h	wchar.h	Description
vfscanf(*stream*, *format*, *arg*)	vfwscanf(*stream*, *format*, *arg*)	Same as fscanf/fwscanf, but extra arguments are given as variable argument list *arg*. (See Section 13.1.20, *stdarg.h*)
vscanf(*format*, *arg*)	vwscanf(*format*, *arg*)	Same as sscanf/swscanf, but extra arguments are given as variable argument list *arg*. (See Section 13.1.20, *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 13.1.20, *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*)
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 13.1.20, *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 13.1.20, *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 13.1.20, *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.

stdio.h	wchar.h	Description
getchar(stdin)	getwchar(stdin)	Reads one character from the stdin stream. Returns the character read or EOF/WEOF on error. Implemented as macro. (*FSS implementation*)
fgets(*s, n, stream)	fgetws(*s, n, stream)	Reads at most the next *n*-1 characters from the *stream* into array *s* until a newline is found. Returns s or NULL or EOF/WEOF on error. (*FSS implementation*)
gets(*s, 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.
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 13.1.6, *errno.h*)

13.1.25. 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 13.1.15, *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`	`long long wcstoll`
`         (*s,**endp,base)`	`                (*s,**endp,base)`
`unsigned long strtoul`	`unsigned long wcstoul`
`         (*s,**endp,base)`	`                (*s,**endp,base)`
`unsigned long long strtoull`	`unsigned long long wcstoull`
`         (*s,**endp,base)`	`                (*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.

Besides these functions, the C166 library supports __near, __far, __huge and __shuge variants of the memory allocation functions.

`__nmalloc`   Allocation in near memory. These functions are default for the near memory
`__ncalloc`   model.
`__nfree`
`__nrealloc`

`__fmalloc`   Allocation in far memory. These functions are default for the far memory model.
`__fcalloc`
`__ffree`
`__frealloc`

`__smalloc`   Allocation in segmented huge memory. These functions are default for the
`__scalloc`   shuge memory model.
`__sfree`
`__srealloc`

`__hmalloc`
`__hcalloc`
`__hfree`
`__hrealloc`
	Allocation in huge memory, *size* is limited to 64 kB. These functions are default for the huge memory model.

`__xhmalloc`
`__xhcalloc`
`__xhfree`
`__xhrealloc`
	Extended allocation in huge memory, *size* is limited to 32-bit.

Based on the memory model the compiler chooses the correct version. For example, if you use `malloc()` in your source and you compile for the near memory model, the compiler uses the function `__nmalloc()`.

Environment communication

`abort()`	Causes abnormal program termination. If the signal SIGABRT is caught, the signal handler may take over control. (See Section 13.1.19, *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)        Compute x/y and x%y in a single operation. X and y have respectively type
ldiv_t ldiv(x,y)      int, long int and long long int. The result is stored in the members
lldiv_t lldiv(x,y)    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 13.1.29, *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 13.1.29, *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 13.1.29, *wchar.h*).

13.1.26. 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`. Additionally the C166 library contains string functions and memory copy functions that use the `__near`, `__far`, `__huge` and `__shuge` memory type qualifiers (see Section 1.3.1, *Memory Type Qualifiers*). These functions all start with a double underscore.

Copying and concatenation functions

string.h	wchar.h	Description
memcpy(*s1,*s2,n)	wmemcpy(*s1,*s2,n)	Copies *n* characters from **s2* into **s1* and returns *s1. If *s1 and *s2 overlap the result is undefined.
memmove(*s1,*s2,n)	wmemmove(*s1,*s2,n)	Same as `memcpy`, but overlapping strings are handled correctly. Returns **s1*.
strcpy(*s1,*s2)	wcscpy(*s1,*s2)	Copies **s2* into **s1* and returns **s1*. If **s1* and **s2* overlap the result is undefined.
strncpy(*s1,*s2,n)	wcsncpy(*s1,*s2,n)	Copies not more than *n* characters from **s2* into **s1* and returns **s1*. If **s1* and **s2* overlap the result is undefined.
strcat(*s1,*s2)	wcscat(*s1,*s2)	Appends a copy of **s2* to **s1* and returns **s1*. If **s1* and **s2* overlap the result is undefined.
strncat(*s1,*s2,n)	wcsncat(*s1,*s2,n)	Appends not more than *n* characters from **s2* to **s1* and returns **s1*. If **s1* and **s2* overlap the result is undefined.

```
__[nfhs]strcpy                              __near, __far, __huge and __shuge variants.
__[nfhs]strncpy
__[nfhs]strcat
__[nfhs]strncat
```

`__memcp[nfhs][nfhs]b`	Copies *n* bytes from one type qualified data to another.
`__memcp[nfhs][nfhs]w`	Copies *n* words from one type qualified data to another.

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 13.1.15, *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*.

```
__[nfhs]strcmp                              __near, __far, __huge and __shuge variants.
__[nfhs]strncmp
__[nfhs]strcoll
__[nfhs]strxfrm
```

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.

string.h	wchar.h	Description
strcspn(*s,*set)	wcscspn(*s,*set)	Searches *s for a sequence of characters *not* specified in *set. Returns the length of the first sequence found.
strpbrk(*s,*set)	wcspbrk(*s,*set)	Same as strspn/wcsspn but returns a pointer to the first character in *s that also is specified in *set.
strstr(*s,*sub)	wcsstr(*s,*sub)	Searches for a substring *sub in *s. Returns a pointer to the first occurrence of *sub in *s.
strtok(*s,*dlm)	wcstok(*s,*dlm)	A sequence of calls to this function breaks the string *s into a sequence of tokens delimited by a character specified in *dlm. The token found in *s is terminated with a null character. Returns a pointer to the first position in *s of the token.
__[nfhs]strchr __[nfhs]strrchr __[nfhs]strspn __[nfhs]strcspn __[nfhs]strpbrk __[nfhs]strstr __[nfhs]strtok		__near, __far, __huge and __shuge variants.

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 13.1.6, *errno.h*)
strlen(*s)	wcslen(*s)	Returns the length of string *s.
__[nfhs]strlen		__near, __far, __huge and __shuge variant of strlen.

13.1.27. 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. When you use the simulator, this function returns the value of the cycle count register CCNT. 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 21 16:15:14 2004\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 13.1.15, *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

%% %

13.1.28. 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*)

13.1.29. wchar.h

Many functions in `wchar.h` represent the wide-character variant of other functions so these are discussed together. (See Section 13.1.24, *stdio.h and wchar.h*, Section 13.1.25, *stdlib.h and wchar.h*, Section 13.1.26, *string.h and wchar.h* and Section 13.1.27, *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 13.1.25, *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 13.1.25, *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 13.1.25, *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*.

13.1.30. wctype.h

Most functions in `wctype.h` represent the wide-character variant of functions declared in `ctype.h` and are discussed in Section 13.1.4, *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 13.1.15, *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 13.1.15, *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")`

13.2. 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 to lengthy for the table.

Function	Not reentrant because
_close	Uses global File System Simulation buffer, _dbg_request
_doflt	Uses I/O functions which modify iob[ ]. See (1).
_doprint	Uses indirect access to static iob[ ] array. See (1).
_doscan	Uses indirect access to iob[ ] and calls ungetc (access to local static ungetc[ ] buffer). See (1).
_Exit	See exit.
_filbuf	Uses iob[ ], which is not reentrant. See (1).
_flsbuf	Uses iob[ ]. See (1).
_getflt	Uses iob[ ]. See (1).
_iob	Defines static iob[ ]. See (1).
_lseek	Uses global File System Simulation buffer, _dbg_request
_open	Uses global File System Simulation buffer, _dbg_request
_read	Uses global File System Simulation buffer, _dbg_request
_unlink	Uses global File System Simulation buffer, _dbg_request
_write	Uses global File System Simulation buffer, _dbg_request
abort	Calls exit
abs labs llabs	-
access	Uses global File System Simulation buffer, _dbg_request
acos acosf acosl	Sets errno.
acosh acoshf acoshl	Sets errno via calls to other functions.
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.

Function	Not reentrant because
calloc __[n|f|s|h|xh]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.
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*)

Function	Not reentrant because
`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.
`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).

Function	Not reentrant because
`fputs fputws`	Uses iob[ ]. See (1).
`fread`	Calls fgetc. See (1).
`free __[n|f|s|h|xh]free`	free uses static buffer management structures. See malloc (5).
`freopen`	Modifies iob[ ]. See (1).
`frexp frexpf frexpl`	-
`fscanf fwscanf`	Uses iob[ ]. See (1)
`fseek`	Uses iob[ ] and calls _lseek. Accesses ungetc[ ] array. See (1).
`fsetpos`	Uses iob[ ] and sets errno. See (1) / (2).
`fstat`	(*Not implemented*)
`ftell`	Uses iob[ ] and sets errno. Calls _lseek. See (1) / (2).
`fwrite`	Uses iob[ ]. See (1).
`getc getwc`	Uses iob[ ]. See (1).
`getchar getwchar`	Uses iob[ ]. See (1).
`getcwd`	Uses global File System Simulation buffer, _dbg_request
`getenv`	Skeleton only.
`gets getws`	Uses iob[ ]. See (1).
`gmtime`	gmtime defines static structure
`hypot hypotf hypotl`	Sets errno via calls to other functions.
`ilogb ilogbf ilogbl`	(*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`	-

Function	Not reentrant because
`isnan`	-
`isnormal`	-
`isprint iswprint`	-
`ispunct iswpunct`	-
`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

Function	Not reentrant because
`lstat`	(*Not implemented*)
`malloc __[n|f|s|h|xh]malloc`	Needs kernel support. See (5).
`mblen`	N.A., skeleton function
`mbrlen`	Sets errno.
`mbrtowc`	Sets errno.
`mbsinit`	-
`mbsrtowcs`	Sets errno.
`mbstowcs`	N.A., skeleton function
`mbtowc`	N.A., skeleton function
`memchr wmemchr`	-
`memcmp wmemcmp`	-
`memcpy wmemcpy`	-
`__memcp[nfhs][nfhs]b`	-
`__memcp[nfhs][nfhs]w`	-
`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

Function	Not reentrant because
`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` `__[n|f|s|h|xh]realloc`	See malloc (5).
`remainder remainderf` `remainderl`	(*Not implemented*)
`remove`	Uses global File System Simulation buffer, _dbg_request
`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` `__[nfsh]strcat`	-
`strchr wcschr` `__[nfsh]strchr`	-
`strcmp wcscmp` `__[nfsh]strcmp`	-

Function	Not reentrant because
strcoll wcscoll __[nfsh]strcoll	-
strcpy wcscpy __[nfsh]strcpy	-
strcspn wcscspn __[nfsh]strcspn	-
strerror	-
strftime wstrftime	-
strlen wcslen __[nfsh]strlen	-
strncat wcsncat __[nfsh]strncat	-
strncmp wcsncmp __[nfsh]strncmp	-
strncpy wcsncpy __[nfsh]strncpy	-
strpbrk wcspbrk __[nfsh]strpbrk	-
strrchr wcsrchr __[nfsh]strrchr	-
strspn wcsspn __[nfsh]strspn	-
strstr wcsstr __[nfsh]strstr	-
strtod wcstod	-
strtof wcstof	-
strtoimax	Sets errno via calls to other functions.
strtok wcstok __[nfsh]strtok	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 __[nfsh]strxfrm	-
system	N.A; skeleton function
tan tanf tanl	Sets errno. See (2).
tanh tanhf tanhl	Sets errno via call to other functions.

Function	Not reentrant because
`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).
`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 into more detail. The numbers at the begin of each paragraph relate to the number references in the table above.

(1) iob structures

The I/O part of the C library is not reentrant by design. This is mainly caused by the static declaration of the `iob[]` array. The functions which use elements of this array access these elements via pointers (`FILE *`).

Building a multi-process system that is created in one link-run is hard to do. The C language scoping rules for external variables make it difficult to create a private copy of the `iob[]` array. Currently, the `iob[]` array has external scope. Thus it is visible in every module involved in one link phase. If these modules comprise several tasks (processes) in a system each of which should have its private copy of `iob[]`, it is apparent that the `iob[]` declaration should be changed. This requires adaptation of the library to the multi-tasking environment. The library modules must use a process identification as an index for determining which `iob[]` array to use. Thus the library is suitable for interfacing to that kernel only.

Another approach for the `iob[]` declaration problem is to declare the array static in one of the modules which create a task. Thus there can be more than one `iob[]` array is the system without having conflicts at link time. This brings several restrictions: Only the module that holds the declaration of the static `iob[]` can use the standard file handles `stdin`, `stdout` and `stderr` (which are the first three entries in `iob[]`). Thus all I/O for these three file handles should be located in one module.

(2) errno declaration

Several functions in the C library set the global variable `errno`. After completion of the function the user program may consult this variable to see if some error occurred. Since most of the functions that set `errno` already have a return type (this is the reason for using `errno`) it is not possible to check successful completion via the return type.

The library routines can set `errno` to the values defined in `errno.h`. See the file `errno.h` for more information.

`errno` can be set to ERR_FORMAT by the print and scan functions in the C library if you specify illegal format strings.

`errno` will never be set to ERR_NOLONG or ERR_NOPOINT since the C library supports long and pointer conversion routines for input and output.

errno can be set to ERANGE by the following functions: `exp()`, `strtol()`, `strtoul()` and `tan()`. These functions may produce results that are out of the valid range for the return type. If so, the result of the function will be the largest representable value for that type and errno is set to ERANGE.

errno is set to EDOM by the following functions: `acos()`, `asin()`, `log()`, `pow()` and `sqrt()`. If the arguments for these functions are out of their valid range (e.g. `sqrt( -1 )`), errno is set to EDOM.

errno can be set to ERR_POS by the file positioning functions `ftell()`, `fsetpos()` and `fgetpos()`.

(3) ungetc

Currently the ungetc buffer is static. For each file entry in the `iob[]` structure array, there is one character available in the buffer to unget a character.

(4) local buffers

`tmpnam()` creates a temporary filename and returns a pointer to a local static buffer. This is according to the ANSI definition. Changing this function such that it creates the name in a user specified buffer requires another calling interface. Thus the function would be no longer portable.

`strtok()` scans through a string and remembers that the string and the position in the string for subsequent calls. This function is not reentrant by design. Making it reentrant requires support of a kernel to store the information on a per process basis.

`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 14. List File Formats

This chapter describes the format of the assembler list file and the linker map file.

14.1. Assembler List File Format

The assembler list file is an additional output file of the assembler that contains information about the generated code. For details on how to generate a list file, see Section 7.5, *Generating a List File*.

The list file consists of a page header and a source listing.

Page header

The page header is repeated on every page:

```
TASKING VX-toolset for C166: Assembler vx.yrz Build nnn SN 00000000
Title                                                         Page 1
May 26 2009 11:42:34

ADDR CODE       CYCLES  LINE SOURCE LINE
```

The first line contains version information. The second line can contain a title which you can specify with the assembler control $TITLE and always contains a page number. The third line contains the invocation date and time. The fourth line is empty and the fifth line contains the headings of the columns for the source listing.

With the assembler controls $[NO]LIST, $PAGELENGTH, $PAGEWIDTH, $[NO]PAGING, 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
                          .
                          .
0004                     25 _16:
0004 E6F2rrrr  2    6    26          movw    r2,#__1_str
0008 DArrrrrr  2    8    27          call    _printf
000C 08F2      2   10    28          addw    r15,#0x2
                          .
                          .
0000                     38          .ds     2
   |   RESERVED
0001
```

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.

14.2. Linker Map File Format

The linker map file is an additional output file of the linker that shows how the linker has mapped the sections and symbols from the various object files (`.obj`) to output sections. The locate part shows the absolute position of each section. External symbols are listed per space with their absolute address, both sorted on symbol and sorted on address. For details on how to generate a map file, see Section 8.9, *Generating a Map File*.

With the linker option **--map-file-format** you can specify which parts of the map file you want to see.

In Eclipse the linker map file (*project*.`mapxml`) is generated in the output directory of the build configuration, usually `Debug` or `Release`. You can open the map file by double-clicking on the file name.
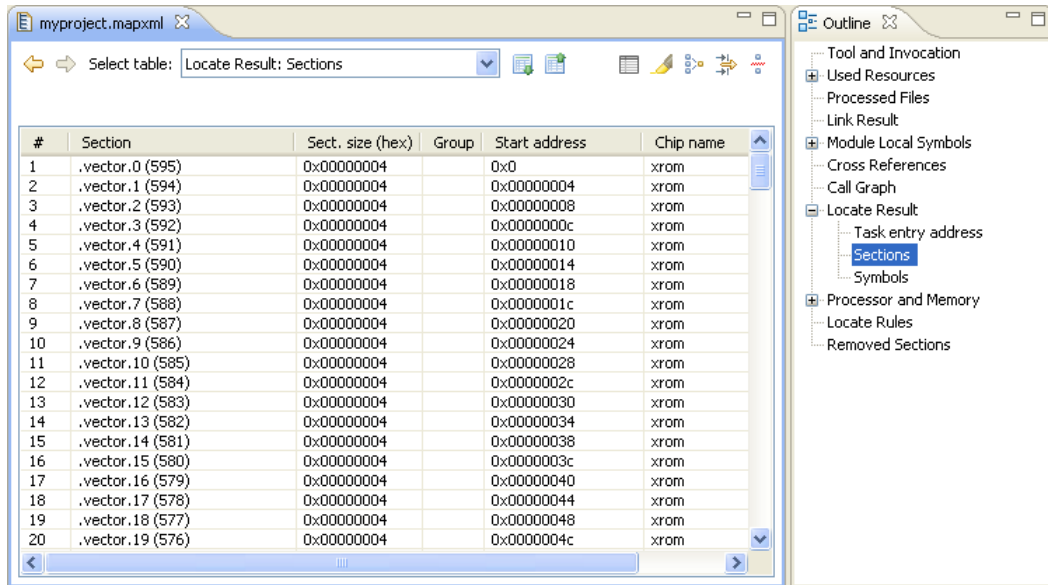
Each page displays a part of the map file. You can use the drop-down list or the Outline view to navigate through the different tables and you can use the following buttons.

Icon	Action	Description
⇦	Back	Goes back one page in the history list.
⇨	Forward	Goes forward one page in the history list.
📄	Next Table	Shows the next table from the drop-down list.
📄	Previous Table	Shows the previous table from the drop-down list.

When you right-click in the view, a popup menu appears (for example, to reset the layout of a table). The meaning of the different parts is:

Tool and Invocation

This part of the map file contains information about the linker, its version header information, binary location and which options are used to call it.

Used Resources

This part of the map file shows the memory usage at memory level and space level. The largest free block of memory (`Largest gap`) is also shown. This part also contains an estimation of the stack usage.

Explanation of the columns:

Memory   The names of the system memory and user memory as defined in the linker script file (`*.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:c16x:near`.
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 (`.obj`) to output sections.

[in] File	The name of an input object file.
[in] Section	A section name and id from the input object file. The number between '( )' uniquely identifies the section.
[in] Size	The size of the input section.
[out] Offset	The offset relative to the start of the output section.
[out] Section	The resulting output section name and id.
[out] Size	The size of the output section.

Module Local Symbols

This part of the map file shows a table for each local scope within an object file. Each table has two columns, one with the symbol name and one with the address of the symbol.

By default this part is not shown in the map file. You have to turn this part on manually with linker option **--map-file-format=+statics** (module local symbols).

Cross References

This part of the map file lists all symbols defined in the object modules and for each symbol the object modules that contain a reference to the symbol are shown. Also, symbols that remain undefined are shown.

Call Graph

This part of the map file contains a schematic overview that shows how (library) functions call each other. To obtain call graph information, the assembly file must contain `.CALLS` directives.

You can click the + or - sign to expand or collapse a single node. Use the ⊞ / ⊟ buttons to expand/collapse all nodes in the call graph.

Icon	Meaning	Description
	Root	This function is the top of the call graph. If there are interrupt handlers, there can be several roots.
	Callee	This function is referenced by several No leaf functions. Right-click on the function and select **Expand all References** to see all functions that reference this function. Select **Back to Caller** to return to the calling function.
	Node	A normal node (function) in the call graph.
	Caller	This function calls a function which is listed separately in the call graph. Right-click on the function and select **Go to Callee** to see the callee. Hover the mouse over the function to see a popup with all callees.

Overlay

This part is empty for the C166.

Locate Result: Sections

This part of the map file shows the absolute position of each section in the absolute object file. It is organized per memory chip and group and sorted on space address. In Eclipse, right-click in the table and select **Configure Columns** to specify which columns you want to see. If you hover the mouse over a section, you get a popup with information about the section. If you select a range of sections, in the Fast View bar (at the bottom) you will see information about the selected range, such as the total size, how many sections are selected and how many gaps are present.

The line number and default sort order.

Section **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.

The following buttons are available in this part of the map file.

Icon	Action	Description
	Configure Section Filter	Opens the Configure Section Filter dialog. Here you can select which sections you want to see in the map file and how.
	Enable Highlighting	All sections that are marked with "Highlight" in the Configure Section Filter dialog will be highlighted in the table.
	Enable Collapsing	All sections that are marked with "Collapse" in the Configure Section Filter dialog will appear collapsed in the table.
	Only Show Matching Lines	All lines that are not part of the selection in the Configure Section Filter dialog will be hidden.
	Show Gaps	Also shows the gaps in the map file. Click the button again to hide the gaps.

Configure Section Filter Dialog

In this dialog you can filter which sections you want to see in the map file and how. Click **Add** to add a new filter. Explanation of the columns and fields:

Highlight	Marks the section as a candidate for highlighting. Turn on **Enable Highlighting** to see the effect.
Color	The highlight color.

Collapse	Marks the section as a candidate for collapsing. Turn on **Enable Collapsing** to see the effect.
Section name	A filter to select a section or group of sections. Wildcards are allowed. Wildcards follow the rules of regular expressions. To get help on which wildcards are supported, press **Ctrl-space**. Click an item in the list for help, double-click to add the wildcard.
Start address	The first address of the section in the address space for this filter.
End address	The last address of the section in the address space for this filter.
Address space	The name of the address space.
Chip name	The name of the memory chip as defined in the linker script file (`*.lsl`) in the `memory` definitions.
Hide gaps smaller than	If gaps are shown in the map file, here you can limit the number of gaps you want to see.

The meaning of the check boxes is the same as the corresponding buttons available in this part of the map file.

Locate Result: Symbols

This part of the map file lists all external symbols.

Address	The absolute address of the symbol in the address space.
Name	The name of the symbol.

Processor and Memory

This part of the map file shows the processor and memory information of the linker script file.

By default this part is not shown in the map file. You have to turn this part on manually with linker option **--map-file-format=+lsl** (processor and memory info). You can print this information to a separate file with linker option **--lsl-dump**.

You can click the + or - sign to expand or collapse a part of the information.

Locate Rules

This part of the map file shows the rules the linker uses to locate sections.

Address space	The names of the address spaces as defined in the linker script file (`*.lsl`). The names are constructed of the `derivative` name followed by a colon ':', the `core` name, another colon ':' and the `space` name.

Type The rule type:

```
ordered/contiguous/clustered/unrestricted
```

Specifies how sections are grouped. By default, a group is 'unrestricted' which means that the linker has total freedom to place the sections of the group in the address space.

```
absolute
```

The section must be located at the address shown in the Properties column.

```
ranged
```

The section must be located anywhere in the address ranges shown in the Properties column; end addresses are not included in the range.

```
page
```

The sections must be located in some address range with a size not larger than shown in the Properties column; the first number is the page size, the second part is the address range restriction within the page.

```
ranged page
```

Both the ranged and the paged restriction apply. In the Properties column the range restriction is listed first, followed by the paged restriction between parenthesis.

```
ballooned
```

After locating all sections, the largest remaining gap in the space is used completely for the stack and/or heap.

Properties The contents depends on the Type column.

Prio The locate priority of the rule. A higher priority value gives a rule precedence over a rule with a lower priority, but only if the two rules have the same type and the same properties. The relative order of rules of different types or different properties is not affected by this priority value. You can set the priority with the priority group attribute in LSL

Sections The sections to which the rule applies;

restrictions between sections are shown in this column:

```
<    ordered
|    contiguous
+    clustered
```

For contiguous sections, the linker uses the section order as shown here. Clustered sections can be located in any relative order.

Removed Sections

This part of the map file shows the sections which are removed from the output file as a result of the optimization option to delete unreferenced sections and or duplicate code or constant data (linker option **--optimize=cxy**).

Section	The name of the section which has been removed.
File	The name of the input object file where the section is removed from.
Library	The name of the library where the object file is part of.
Symbol	The symbols that were present in the section.
Reason	The reason why the section has been removed. This can be because the section is unreferenced or duplicated.

Chapter 15. Object File Formats

This chapter describes the format of several object files.

15.1. ELF/DWARF Object Format

The TASKING VX-toolset for C166 by default produces objects in the ELF/DWARF 3 format.

For a complete description of the ELF format, please refer to the *Tool Interface Standard (TIS)*.

For a complete description of the DWARF format, please refer to the *DWARF Debugging Information Format Version 3*. See http://dwarfstd.org/

The implementation of the ELF object format and the DWARF 3 debug information for the TASKING VX-toolset for C166 is described in the TASKING C166 ELF/DWARF Application Binary Interface (EDABI). You can download the latest PDF document from the Altium website at: http://www.altium.com/TASKING/support/c166/

15.2. Intel Hex Record Format

Intel Hex records describe the hexadecimal object file format for 8-bit, 16-bit and 32-bit microprocessors. The hexadecimal object file is an ASCII representation of an absolute binary object file. There are six different types of records:

- Data Record (8-, 16, or 32-bit formats)

- End of File Record (8-, 16, or 32-bit formats)

- Extended Segment Address Record (16, or 32-bit formats)

- Start Segment Address Record (16, or 32-bit formats)

- Extended Linear Address Record (32-bit format only)

- Start Linear Address Record (32-bit format only)

To generate an Intel Hex output file:

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

    The Properties dialog appears.

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

    In the right pane the Settings appear.

3. On the Tool Settings tab, select **Linker » Output Format**.

4. Enable the option **Generate Intel Hex format file**.

5. (Optional) Specify the **Size of addresses (in bytes) for Intel Hex records**.

6. (Optional) Enable or disable the option **Emit start address record**.

By default the linker generates records in the 32-bit format (4-byte addresses).

General Record Format

In the output file, the record format is:

:		*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:

```
:040000050000048073
 | |   | |         |_ 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
```

15.3. Motorola S-Record Format

To generate a Motorola S-record output file:

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

   The Properties dialog appears.

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

   In the right pane the Settings appear.

3. On the Tool Settings tab, select **Linker » Output Format**.

4. Enable the option **Generate S-records file**.

5. (Optional) Specify the **Size of addresses (in bytes) for Motorola S records**.

By default, the linker produces output in Motorola S-record format with three types of S-records (4-byte addresses): S0, S3 and S7. Depending on the size of addresses you can force other types of S-records. They have the following layout:

S0 - record

S0	*length*	0000	*comment*	*checksum*

A linker generated S-record file starts with an S0 record with the following contents:

```
        l k 1 6 6
S00800006C6B31363683
```

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:

```
S7050000048076
   | |         |_checksum
   | |_ address
   |_ length
```

Chapter 16. 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.

16.1. 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 16.4, *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 16.5, *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 16.6, *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 16.6.3, *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 16.8, *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
}
```

16.2. 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.

16.2.1. 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.

16.2.2. 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 '`//`'.

16.2.3. 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.

16.2.4. 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
```

16.2.5. 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 your 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.

16.2.6. LSL Definitions in the Linker Script File

$$description \qquad ::= \text{<}definition\text{>}^{>=1}$$

$$definition \qquad ::= architecture\_definition$$
$$| \; derivative\_definition$$
$$| \; board\_spec$$
$$| \; section\_definition$$
$$| \; section\_setup$$

- At least one *architecture_definition* must be present in the LSL file.

16.2.7. Memory and Bus Definitions

$$mem\_def \qquad ::= \textbf{memory} \; mem\_name \; \text{<}tag\_attr\text{>}^{0|1} \; \{ \;\; \text{<}mem\_descr \; \textbf{;}\text{>}^{>=0} \; \}$$

- A *mem_def* defines a memory with the *mem_name* as a unique name.

$$mem\_descr \qquad ::= \textbf{type =} \; \text{<}\textbf{reserved}\text{>}^{0|1} \; mem\_type$$
$$| \; \textbf{mau =} \; expr$$
$$| \; \textbf{size =} \; expr$$
$$| \; \textbf{speed =} \; number$$

```
                        |  fill <= fill_values>^{0|1}
                        |  mapping
```

- A *mem_def* contains exactly one **type** statement.

- A *mem_def* contains exactly one **mau** statement (non-zero size).

- A *mem_def* contains exactly one **size** statement.

- A *mem_def* contains zero or one **speed** statement (if absent, the default speed value is 1).

- A *mem_def* contains zero or one **fill** statement.

- A *mem_def* contains at least one *mapping*

```
mem_type          ::= rom          // attrs = rx
                    | ram          // attrs = rw
                    | nvram        // attrs = rwx
                    | 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.

16.2.8. 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
```

$$vector\_def \quad ::= \textbf{vector (} vector\_spec \textbf{ <, } vector\_spec\textbf{>}^{>=0} \textbf{);}$$

```
vector_spec         ::= id = vector_id_spec
                      | fill = vector_value
                      | optional
                      | tag
```

$$vector\_id\_spec \quad ::= number$$
$$\quad | \textbf{ [} range \textbf{] <, [} range \textbf{]>}^{>=0}$$

```
vector_value        ::= symbol_name
```
$$\quad | \textbf{ [} number \textbf{ <, } number\textbf{>}^{>=0} \textbf{]}$$
$$\quad | \textbf{ loop <[} expr \textbf{]>}^{0|1}$$

$$reserved\_range \quad ::= \textbf{reserved <}tag\_attr\textbf{>}^{0|1} expr \textbf{ .. } expr \textbf{ ;}$$

• The end address is not part of the range.

$$endianness\_def \quad ::= \textbf{endianness \{ <}endianness\_type\textbf{;>}^{>=1} \textbf{ \}}$$

```
endianness_type     ::= big
                      | little
```

16.2.9. Derivative Definition

```
derivative_definition
```
$$\quad ::= \textbf{derivative } derivative\_name$$
$$\quad \textbf{<(} parameter\_list \textbf{)>}^{0|1}$$
$$\quad \textbf{<extends } derivative\_name$$
$$\quad \textbf{<(} argument\_list \textbf{)>}^{0|1} \textbf{ >}^{0|1}$$
$$\quad \textbf{\{ <}derivative\_spec\textbf{>}^{>=0} \textbf{ \}}$$

• 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*.

16.2.10. 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.

16.2.11. 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 ;
```

16.3. 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.

16.4. 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
```

16.4.1. 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
}
```

16.4.2. 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 16.4.4, *Mappings*.

```
bus bus_name
{
    mau = 8;
    width = 8;
    map ( map_description );
}
```

16.4.3. 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 16.8.2, *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 16.4.4, *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 16.8.3, *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 16.8.3, *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 16.8, *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 16.8.2, *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 16.8.3, *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 );
}
```

16.4.4. 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 Flash4
{
  mau   = 8;
  type  = rom;
  size  = 64k;
  map not_cached (dest=bus:xc16x:address, dest_offset=0xD00000, size=64k);
  map cached     (dest=bus:xc16x:address, dest_offset=0x900000, size=64k,
                                          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.

16.5. 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
   map
section_layout
section_setup

   map
      dest          bus   space
```

```
        dest_dbits
        dest_offset
        size
        src_dbits
        src_offset
        reserved
```

16.5.1. 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
}
```

16.5.2. 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);
}
```

16.5.3. 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 16.6.3, *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 16.8.2, *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 16.4.4, *Mappings*.

- The optional **`fill`** field contains a bit pattern that the linker writes to all memory addresses that remain unoccupied during the locate process. The result of the expression, or list of expressions, is used as values to write to memory, each in MAU.

```
memory mem_name
{
    type = rom;
    mau = 8;
    fill = 0xaa;
    size = 64k;
    speed = 2;
    map map_name ( map_description );
}
```

With the **bus** keyword you define a bus in a derivative definition. Buses are described in Section 16.4.2, *Defining Internal Buses*.

16.6. 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
    map

    map
        dest         bus  space
        dest_dbits
        dest_offset
        size
        src_dbits
        src_offset
        reserved
```

16.6.1. 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
}
```

16.6.2. 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);
}
```

16.6.3. Defining External Memory and Buses

It is common to define external memory (off-chip) and external buses at the global scope (outside any enclosing definition). Internal memory (on-chip memory) is usually defined in the scope of a derivative definition.

With the keyword **memory** you define physical memory that is present on the target board. The memory name is used to identify the memory and does not conflict with other identifiers. If you define memory parts in the LSL file, only the memory defined in these parts is used for placing sections.

If no external memory is defined in the LSL file and if the linker option to allocate memory on demand is set then the linker will assume that all virtual addresses are mapped on physical memory. You can override this behavior by specifying one or more memory definitions.

```
memory mem_name
{
    type = rom;
    mau = 8;
```

```
   fill = 0xaa;
   size = 64k;
   speed = 2;
   map map_name ( map_description );
}
```

For a description of the keywords, see Section 16.5.3, *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 16.4.2, *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.

16.7. 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
```

16.7.1. 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 16.4.3, *Defining Address Spaces* for details on the keywords **stack**, **heap**, **vector_table** and **reserved**.

16.8. 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
```

16.8.1. 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.

16.8.2. 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 16.8.3, *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.

When you use wildcards, the linker has the possibility to skip some sections in the selection process when it is obvious you did not want to select these. For example, inlined vector sections, but also a start section already having an absolute start address.

For example, when you specify restrictions on code sections excluding vector handler code, you should use a wildcard to select the code sections. `select "code*";` will not select vector handler code sections, whereas `select "code";` does.

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 16.4.3, *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";
}
```

16.8.3. 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 16.4.3, *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
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.

16.8.4. 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_copy_table" := "__lc_ub_table";
    // when the symbol __lc_copy_table occurs as an undefined reference
    // in an object file, the linker generates a copy table
}
```

16.8.5. 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 17. Debug Target Configuration Files

DTC files (Debug Target Configuration files) define all possible configurations for a debug target. A debug target can be target hardware such as an evaluation board or a simulator. The DTC files are used by Eclipse to configure the project and the debugger. The information is used by the property pages and the debug configuration. DTC files are located in the `etc` directory of the installed product and use `.dtc` as filename suffix.

17.1. Custom Board Support

When you need support for a custom board and the board requires a different configuration than those that are in the product, it is necessary to create a dedicated DTC file.

To add a custom board

1. From the `etc` directory of the product, make a copy of a `.dtc` file and put it in your project directory (in the current workspace).

   In Eclipse, the DTC file should now be visible as part of your project.

2. Edit the file and give it a name that reflects the custom board.

The Import Board Configuration wizard in Eclipse adds DTC files that are present in your current project to the list of available target boards.

Syntax of a DTC file

DTC files are XML files and use the XML Schema file `dtc.xsd`, also present in the `etc` directory of the installed product.

Inspect the DTC XML schema file `dtc.xsd` for a description of the allowed elements and the available attributes. Use a delivered `.dtc` file as a starting point for creating a custom board specification.

Basically a DTC file consists of the definition of the debug target (`debugTarget` element) which embodies one or more configurations (`configuration` element) and one or more communication methods (`communicationMethod` element). The Import Board Configuration wizard in Eclipse reflects the structure of the DTC file. The elements that determine the settings that are applied by the wizard, can be found at any level in the DTC file. The wizard will apply all elements that are within the path to the selected configuration. This is best explained by an example of a DTC file with the following basic layout:

```
debugTarget: Infineon XC16x Starter Kit Board
    lsl
    configuration: Single Chip
        lsl
    configuration: External memory, multiplexed bus
        lsl
    configuration: External memory, non-multiplexed bus
        lsl
    configuration: External memory, multiplexed bus, debug monitor
        lsl
```

```
configuration: External memory, non-multiplexed bus, debug monitor
    lsl
communicationMethod: DAS over MiniWiggler

communicationMethod: DAS over USB-Wiggler

communicationMethod: ROM/RAM Monitor

lsl
```

If, in the Import Board Configuration wizard in Eclipse, you set the board configuration to "External memory, multiplexed bus, debug monitor", the wizard puts the following LSL parts into the project's LSL file in this order:

- the lsl part under the `debugTarget` element

- the lsl part under the `configuration` "External memory, multiplexed bus, debug monitor"

- the lsl part in the `debugTarget` element at the end of the DTC file

The same applies to all other elements that determine the underlying settings.

DTC macros in LSL

To protect the Import Board Configuration wizard from changing the LSL file, you can protect the LSL file by adding the macro `__DTC_IGNORE`. This can be useful for projects that need the same LSL file, but still need to run on different target boards.

```
#define __DTC_IGNORE
```

The following DTC macros can be present in the LSL file:

LSL Define	Description
`__DTC_IGNORE`	If defined, protects the LSL file against changes by the Import Board Configuration wizard.
`__DTC_START` `__DTC_END`	The LSL part that is between these macros can be replaced by LSL text from the DTC file. If the macros are not present in the LSL file, the Import Board Configuration wizard will add them.

17.2. Description of DTC Elements and Attributes

The following table contains a description of the elements as can be found in the XML schema file `dtc.xsd`. For each element a list of allowed elements is listed and the available attributes are described.

Element / Attribute	Description	Allowed Elements
debugTarget	The debug target.	flashChips, lsl, communicationMethod, configuration, def, processor, resource, initialize
name	The name of the configuration.	
manufacturer	The manufacturer of the debug target.	
processor	Defines a processor that can be present on the debug target. Multiple processor definitions are allowed. The user should select the actual processor on the debug target.	-
name	A descriptive name of the processor derivative.	
cpu	Defines the CPU name, as for example supplied with the option **--cpu** of the C compiler.	
communicationMethod	Defines a communication method. A communication method is the channel that is used to communicate with the target, for example a ROM monitor via RS232.	ref, resource, initialize
name	A descriptive name of the communication method.	
debugInstrument	The debug instrument DLL/Shared library file to be used for this communication method. Do not supply a path or a filename suffix.	
gdiMethod	This is the method used for communication. Allowed values: rs232, tcpip, can, none	
def	Defines a set of elements as a macro. The macro can be expanded using the ref element.	lsl, resource, initialize, ref, configuration, flashMonitor
id	The macro name.	
resource	Defines a resource definition that can be used by Eclipse, the debugger or by the debug instrument.	-
id	The identifier name used by the debugger or debug instrument to retrieve the value.	
value	The value assigned to the resource.	
ref	Reference to a macro defined with a def element. The elements contained in the def element with the same name will be expanded at the location of the ref. Multiple ref elements to the same def are allowed.	-
id	The name of the referenced macro.	

Element / Attribute	Description	Allowed Elements
`configuration`	Defines a configuration.	ref, initialize, resource, lsl, information, flashMonitor
`name`	The descriptive name of the configuration.	
`initialize`	This element defines an initialization expression. Each initialize element contains a `resourceId` attribute. If the debug instrument requests this resource, the debugger will compose a string from all initialize elements with the same `resourceId`. The debug instrument can use this string to initialize registers by passing it to the debugger as an expression to be evaluated.	-
`resourceId`	The name of the resource to be used.	
`name`	The name of the register to be initialized.	
`value`	When the `cstart` attribute is false, this is the value to be used, otherwise, it is the default value when using this configuration. It will be used by the startup code editor to set the default register values.	
`cstart`	A boolean value. If true, the debugger should ask the C startup code editor for the value, otherwise the contents of the value attribute is used. The default value is true.	
`flashMonitor`	This element specifies the flash programming monitor to be used for this configuration.	-
`monitor`	Filename of the monitor, usually an Intel Hex or S-Record file.	
`workspaceAddress`	The address of the workspace of the flash programming monitor.	
`flashSectorBufferSize`	Specifies the buffer size for buffering a flash sector.	
`chip`	This element defines a flash chip. It must be used by the flash properties page to add it on request to the list of flash chips.	debugTarget
`vendor`	The vendor of this flash chip.	
`chip`	The name of the chip.	
`width`	The width of the chip in bits.	
`chips`	The number of chips in parallel.	
`baseAddress`	The base address of the chip.	
`chipSize`	The size of the chip in bytes.	
`flashChips`	Specify a list of flash chips that can be available on this debug target.	chip

Element / Attribute	Description	Allowed Elements
`information`	The description given in this element will be used in the User Interface to inform the user about the configuration.	line
`lsl`	Defines LSL pieces belonging to the configuration part. The LSL text must be defined between the start and end tag of this element. All LSL texts of the active selection will be placed in the project's LSL file.	-
`line`	A single line within an `information` element. It can be used to format the text with line breaks.	-

17.3. Special Resource Identifiers

The following resource IDs are available in the TASKING VX-toolset for C166:

DAS debug instrument (DI): gdi2das

Resource Name	Description	Possible Values
`AccessPort`	The port used to connect to the wiggler.	JTAG0, JTAG1, USB
`DASserver`	The DAS Server used for communication.	JTAG JDRV LPT JTAG over USB Box JTAG over USB Chip UDAS
`DasTimeOut`	The timeout value for communication with the DAS server in milliseconds. The default is 0x4000.	
`RegisterFile`	The core register file that is used by the debug instrument. This is usually "ocdsext2mac.dat", because this is the only active core with OCDS support.	
`TerminateServer`	Terminate the DAS server when the session is closed.	0, 1

ROM/RAM monitor debug instrument: dieva166

Resource Name	Description	Possible Values
`boot`	The bootstrap loader S-Record or Hex file for this board.	
`bslack`	A comma separated list of allowed bootstrap acknowledge bytes.	
`monitor`	The monitor S-Record or Hex file for this board.	
`driver`	Driver used for CAN access, for example: pcan.	

Resource Name	Description	Possible Values
`reset_period`	The period a reset signal is applied (in milliseconds).	
`rs232_bootstrap_level`	The level that makes the bootstrap active.	0, 1
`rs232_bootstrap_pin`	The RS232 pin that controls the bootstrap mode.	RTS, DTR
`rs232_reset_level`	The level that makes the reset active.	0, 1
`rs232_reset_pin`	The RS232 pin where the reset is connected to.	RTS, DTR
`rs232_bootstrap_hold_level`	The level that holds the bootstrap mode.	0, 1

Simulator debug instrument: disim166

Resource Name	Description
`psm_dll_name`	The DLL name for peripheral simulation. For example: psm166

Eclipse

Resource Name	Description
`__DPP0_ADDR`, `__DPP1_ADDR`, `__DPP2_ADDR`	The address assigned to the DPPs in the LSL file.
`baudrates`	Comma separated list of possible baud rates. The wizard will only show these baud rates.
`defaultBaudrate`	The baud rate set as default in the wizard.
`flash_note`	A note text that can be shown with the flash properties.
`vectortable.run_addr`	The address set as start address of the interrupt vector table in the LSL file.

17.4. Initialize Elements

The `initialize` elements are used to initialize SFRs at startup. This is also done using a resource of the debug instrument. The following resource Ids exist for the RAM/ROM monitor debug instrument (dieva166):

Resource Name	Description
`einit`	Initialize SFR before the EINIT instruction in the monitor.
`init`	Initialize SFR after the EINIT instruction in the monitor.

The DAS debug instrument (gdi2das) has the same resource Ids, but there is no difference between them.

Chapter 18. CPU Problem Bypasses and Checks

Infineon Components and STMicroelectronics regularly publish microcontroller errata sheets for reporting both functional problems and deviations from the electrical and timing specifications.

For some of these functional problems in the microcontroller itself, the TASKING C166 compiler provides workarounds. In fact these are software workarounds for hardware problems.

This chapter lists a summary of functional problems which can be bypassed by the compiler toolset. Please refer to the Infineon / STMicroelectronics errata sheets for the CPU step you are using, to verify if you need to use one of these bypasses.

To set a CPU bypass or check

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

    The Properties dialog appears.

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

    In the right pane the Processor page appears.

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

    *The **CPU problem bypasses and checks** box shows the available workarounds/checks available for the selected processor.*

4. (Optional) Select **Show all CPU problem bypasses and checks**.

5. Click **Select All** or select one or more individual options.

Overview of the CPU problem bypasses and checks

The following table contains an overview of the silicon bug numbers you can provide to the compiler and assembler option **--silicon-bug**. WA means a workaround by the compiler, CK means a check by the assembler.

Number	Description	Workaround	Check	CPU
1	BUS.18 -- JMPR at jump target address	WA	CK	C161OR C161RI C161V C163 C164CL C164SI C167CR
2	CPU 1R006 -- CPU hangup with MOV(B) Rn,[Rm+#data16]	WA	CK	
3	CPU.3 -- MOV(B) Rn,[Rm+#data16] as the last instruction in an extend sequence	WA	CK	
4	CPU.11 -- Interrupted multiply	WA		

Number	Description	Workaround	Check	CPU
5	CPU.16 -- MOVB [Rn],mem	WA	CK	
6	CPU.18/CPU.2 -- Interrupted multiply and divide instructions	WA	CK	
7	CPU.21 -- Incorrect result of BFLDL/BFLDH after a write to internal RAM	WA	CK	C161CS C161K C161O C161PI C164CL C164SI C167CR C167CS C167CS40
8	CPU_SEGPEC -- PEC interrupt after CP or SRCPx update may use incorrect source pointer value	WA	CK	C166Sv1
9	CR105619 -- Phantom interrupt occurs if Software Trap is cancelled	WA	CK	C166Sv1
10	CR105893 -- Interrupted division corrupted by division in interrupt service routine	WA	CK	C166Sv1
11	CR105981 -- JBC and JNBS with op1 a DPRAM operand (bit addressable) do not work	WA	CK	C166Sv1
12	CR107092 -- Extended sequences not properly handled with conditional jumps	WA	CK	C166Sv1
13	CR108219 -- Old value of SP used when second operand of SCXT points to SP		CK	C166Sv1
14	CR108309 -- MDL access immediately after a DIV causes wrong PSW values	WA	CK	C166Sv1
15	CR108342 -- Lost interrupt while executing RETI instruction	WA	CK	C166Sv1
16	CR108361 -- Incorrect (E)SFR address calculated for RETP as last instruction in extend sequence		CK	
17	CR108400 -- Broken program flow after not taken JMPR/JMPA instruction	WA	CK	C166Sv1
18	CR108904 -- DIV/MUL interrupted by PEC when the previous instruction writes in MDL/MDH	WA	CK	C166Sv1
19	Kfm_BR03 -- Pipeline conflict after CoSTORE	WA		
20	LONDON.1 -- Breakpoint before JMPI/CALLI	WA	CK	C166Sv2_0BO
21	LONDON.1751 -- Write to core SFR while DIV[L][U] executes	WA	CK	C166Sv2_0BO
22	LONDON RETP -- Problem with RETP on CPU SFRs		CK	
25	ST_BUS.1 -- JMPS followed by PEC transfer	WA	CK	
26	CPU_MOVB_JB -- Jump-bit wrongly executed after byte manipulation instructions	WA	CK	C166Sv1

Number	Description	Workaround	Check	CPU
	CPU.22 -- Z Flag after PUSH and PCALL The TASKING VX-toolset for C166 never generates this problem.			

BUS.18 -- JMPR at jump target address

Infineon / STMicroelectronics reference

BUS.18

Command line option

`--silicon-bug=1`

Libraries

lib\p1*.lib

Description

If a PEC transfer occurs immediately after a JMPR instruction the program counter can have a wrong value. There are many other requirements before this actually happens, among others the JMPR has to be reached by a jump instruction.

When you use the assembler option **--silicon-bug=1**, the assembler issues a warning when the BUS.18 problem is present.

CPU 1R006 -- CPU hangup with MOV(B) Rn,[Rm+#data16]

Infineon reference

CPU1R006

Command line option

`--silicon-bug=2`

Libraries

`lib\p1\*.lib`

Description

The opcode `MOV (B) Rn, [Rm+#data16]` can cause the CPU to hang. The problem is encountered under the following conditions:

- `[Rm+#data16]` is used to address the source operand

- `[Rm+#data16]` points to the program memory

- a hold cycle has to be generated by the `ir_ready` signal at the beginning of the operand fetch cycle

Since the compiler is unaware of the actual location the source operand `[Rm+#data16]` refers to, the generation of this addressing mode is completely suppressed.

When you use the assembler option **--silicon-bug=2**, the assembler issues a fatal error when the CPU1R006 problem is present.

CPU.3 -- MOV(B) Rn,[Rm+#data16] as the last instruction in an extend sequence

Infineon reference

CPU.3

Command line option

`--silicon-bug=3`

Libraries

`lib\p1\*.lib`

Description

On older C167 derivatives the last instruction in an extend sequence will use a DPP translation instead of the page or segment number supplied with the extend instruction (EXT*xx*). This problem occurs only when the last instruction of this extend instruction uses the addressing mode `Rn,[Rm+#data16]` or `[Rm+#data16],Rn`. When you use the C compiler option **--silicon-bug=3** the compiler will lengthen the extend sequence with one instruction when it generates an instruction using this addressing mode.

When you use the assembler option **--silicon-bug=3**, the assembler issues a warning when this instruction is found at the end of EXTP, EXTPR, EXTS or EXTSR sequences.

CPU.11 -- Interrupted multiply

Infineon reference

CPU.11

Command line option

`--silicon-bug=4`

Libraries

`lib\p1\*.lib`

Description

Use this solution where failures occur for interrupts during the MUL and MULU instructions. Multiply operations are protected inline using ATOMIC instructions. In some cases, an additional NOP might be generated after the multiply instruction.

You should also link libraries in which the divide operations are protected. The libraries in the directory `lib\p1` also have the divide protected against interrupts, but can be used safely to bypass this CPU problem.

CPU.16 -- MOVB [Rn],mem

Infineon reference

CPU.16

Command line option

`--silicon-bug=5`

Libraries

`lib\p1\*.lib`

Description

When the `MOVB[Rn],mem` instruction is executed, where (a) `mem` specifies a direct 16-bit byte operand address in the internal ROM/Flash memory, and (b) `[Rn]` points to an even byte address, while the contents of the word which includes the byte addressed by `mem` is odd, or `[Rn]` points to an odd byte address, while the contents of the word which includes the bytes addressed by `mem` is even, the following problem occurs:

- when `[Rn]` points to external memory or to the X-Peripheral (XRAM, CAN, etc.) address space, the data value which is written back is always 0x00.

- when `[Rn]` points to the internal RAM or SFR/ESFR address space, (a) the (correct) data value `[mem]` is written to `[Rn]+1`, i.e. to the odd byte address of the selected word in case `[Rn]` points to an even byte address, (b) the (correct) data value `[mem]` is written to `[Rn]-1`, i.e. to the even byte address of the selected word in case `[Rn]` points to an odd byte address.

Since internal ROM/Flash/OTP data is referred to as 'const' data, the compiler will prevent generating the `MOVB [Rn],mem` instruction when even 'const' objects are accessed. The compiler is unaware of the exact location of these objects which is determined at locate time.

When you use the assembler option **--silicon-bug=5**, the assembler issues a fatal error when the CPU.16 problem is present.

CPU.18/CPU.2 -- Interrupted multiply and divide instructions

Infineon reference

CPU.18 and CPU.2

Command line option

`--silicon-bug=6`

Libraries

lib\p1*.lib

Description

Use this solution where failures occur for interrupts during the MUL, MULU, DIV, DIVU, DIVL and DIVLU instructions. Multiply and divide operations are protected inline using ATOMIC instructions. In some cases, an additional NOP might be generated after the multiply or divide instruction.

C compiler option **--silicon-bug=6** is a workaround for many MUL/DIV problems. Besides CPU.18 it fixes CPU.2 and CPU.11.

When you use this option you should also link the libraries in which the multiply and divide operations are protected.

When you use the assembler option **--silicon-bug=6**, the assembler issues a warning whenever a MUL or DIV is encountered that is not protected by an ATOMIC sequence.

CPU.21 -- Incorrect result of BFLDL/BFLDH after a write to internal RAM

Infineon / STMicroelectronics reference

CPU.21

Command line option

`--silicon-bug=7`

Libraries

`lib\p1\*.lib`

Description

The result of a BFLDL/BFLDH instruction may be incorrect after a write to internal RAM. This only happens under very specific circumstances.

When you use the assembler option **--silicon-bug=7**, the assembler checks for the CPU.21 silicon problem and issues warnings and errors:

- an error when the previous operation writes to a register (including post increment, pre increment, post decrement and pre decrement) whose 8 bit address equals the appropriate field in the BFLDx operation.

- a warning if the previous operation writes to a register and the BFLDx instruction has a relocatable value in the concerned field.

- a warning if the previous instruction uses indirect addressing or executes an implicit stack write a warning if the previous instruction writes to IRAM and the BFLDx field is relocatable or larger than 0xEF.

- a warning if the previous instruction writes to bit addressable IRAM (including writing to a register) and the BFLDx field is relocatable or smaller than 0xF0.

- a warning if the BFLDx instruction is not protected by ATOMIC, EXTR, EXTP, EXTPR, EXTS or EXTSR, which means a PEC transfer may occur just before the execution of BFLDx.

- a warning after any PCALL, because such routines normally use the RETP instruction, which could cause a problem a warning after any RETP, because a BFLDx could follow directly, which could in turn cause a problem.

CPU.22 -- Z Flag after PUSH and PCALL

Infineon / STMicroelectronics reference

CPU.22

Command line option

Not available, because problem does not occur.

Description

The TASKING VX-toolset for C166 never generates a PCALL instruction, nor is it used in the libraries. The PUSH instruction is only used in the entry of an interrupt frame, and sometimes on exit of normal functions. The zero flag is not a parameter or return value, so this does not give any problems.

Since code generated by the C166 compiler is not affected, analysis and workarounds are only required for program parts written in assembly, or instruction sequences inserted via inline assembly.

Workaround (for program parts written in assembly)

Do not evaluate the status of the Z flag generated by a PUSH or PCALL instruction. Instead, insert an instruction that correctly updates the PSW flags, e.g.

```
    PUSH reg
    CMP reg, #0 ; updates PSW flags
    ; note: CMP additionally modifies the C and V flags,
    ; while PUSH or MOV leaves them unaffected
    JMPR cc_Z, label_1 ; implicitly tests Z flag
```

or

```
    PCALL reg, procedure_1
    ...
procedure_1:
    MOV ONES, reg          ; updates PSW flags
    JMPR cc_NET, label_1  ; implicitly tests flags Z and E
```

Whether or not an instruction following `PUSH reg` or `PCALL reg, rel` actually causes a problem depends on the program context. In most cases, it will be sufficient to just analyze the instruction following PUSH or PCALL. In case of PCALL, this is the instruction at the call target address.

CPU_SEGPEC -- PEC interrupt after CP or SRCPx update may use incorrect source pointer value

Infineon reference

CPU_SEGPEC

Command line option

`--silicon-bug=8`

Libraries

`lib\p1\*.lib`

Description

The Infineon EWGold Lite core can have a problem when PEC interrupts arrive to close together. This can cause a wrong SRCPx source value to be used for the PEC transfer. The problem also occurs when the context pointer register CP is explicitly modified.

When you use the assembler option **--silicon-bug=8**, the assembler issues an error when the CPU_SEGPEC problem is present.

Workaround

Guard the offending instructions against PEC interrupts through an EXTEND sequence. For explicit CP modifications, the extend sequence needs to be 3 instructions at least, for the SRCPx modifications, the sequence needs to be 2 instructions at least.

```
SCXT CP, #12  ;; possible problem, error is generated
ATOMIC #3
MOV CP, R1    ;; properly guarded
MOV SRCP0, R1 ;; properly guarded
ADD SRCP0, R1 ;; possible problem, error is generated
```

CR105619 -- Phantom interrupt occurs if Software Trap is cancelled

Infineon reference

CR105619

Command line option

`--silicon-bug=9`

Libraries

`lib\p1\*.lib`

Description

The last regularly executed interrupt is injected again if a software trap is canceled and at the same time a real interrupt occurs. A sequence where this problem occurs is the following:

```
BMOV    R13.1,0FD10h.1
TRAP    #010h
```

Due to the previous operation the TRAP is canceled and at the same time a real interrupt occurs. As a result of this, the last previously executed interrupt is injected and then the real interrupt is injected too (if its priority is high enough).

Conditions for canceling a software TRAP are:

• previous instruction changes SP (explicitly)

• previous instruction changes PSW (implicit or explicitly)

• OCDS/hardware triggers are generated on the TRAP instruction

When you use the assembler option **--silicon-bug=9**, the assembler issues errors if it finds TRAP operations directly preceded by SP or PSW modifying instructions. It also issues warnings on level 2 for cases where this problem could occur, for example at labels or after RETP or JBC instructions.

Workaround

Do not cancel a software trap by inserting a NOP before a TRAP instruction.

CR105893 -- Interrupted division corrupted by division in interrupt service routine

Infineon reference

CR105893

Command line option

`--silicon-bug=10`

Libraries

`lib\p1\*.lib`

Description

In the first states of a division several internal control signals are set that are used in later states of the division. If a division is interrupted and in the interrupt service routine (ISR) another division is executed, it overrides the old internal values. After the return the interrupted division proceeds with the (probably wrong) internal states of the last division. The affected internal signals are `dividend_sign`, `divisisor_sign` and `mdl_0`. The first two bits represent the operand signs (=Bit 15). `mdl_0` is set if MDL is 0xFFFF.

When you use the assembler option **--silicon-bug=10**, the assembler issues a warning if an unprotected DIV is found.

Workaround

Do not interrupt divisions, for example by using an ATOMIC sequence.

CR105981 -- JBC and JNBS with op1 a DPRAM operand (bit addressable) do not work

Infineon reference

CR105981

Command line option

`--silicon-bug=11`

Libraries

`lib\p1\*.lib`

Description

The DPRAM address (corresponding to `op1`) is written back with wrong data. This happens even if the jump is not taken.

> Note that these instructions work properly for GPR operands and SFR operands.

When you use the assembler option **--silicon-bug=11**, the assembler accepts the compiler workaround silently and issues no warning.

Workaround

The JBC and JNBS instructions are protected by ATOMIC instructions, unless the first operand is a GPR.

CR107092 -- Extended sequences not properly handled with conditional jumps

Infineon reference

CR107092

Command line option

`--silicon-bug=12`

Libraries

`lib\p1\*.lib`

Description

Affected are the instructions EXTR, EXTP, EXTPR, EXTS, EXTSR and ATOMIC since the responsible code generates the control signals for all these instructions, however, the effects will differ.

Example:

```
            EXTR    #1
            JB      DP1H.6, JMP_TARGET    ; taken jump
            MOV     MDC, #0000Fh
            MOV     MDH, MDL
            CALL    never_reached
JMP_TARGET:
            MOV     MDC, #0000Fh
            MOV     MDH, MDL
```

In this example the jump is correctly executed and taken. However, the control signal for the extended register sequence is not reset. So, at `JMP_TARGET` the extend sequence is still effective. This means that the move instruction is extended and instead of writing to the SFR `MDC` (`FF0Eh`) the move instruction writes to address `F10Eh`, an ESFR address.

The bug occurs with taken conditional jumps only, since they are executed as two cycle commands and therefore re-injected in the pipeline. If the jump is not taken or unconditional, the sequence above will work properly, since these jumps are executed as single cycle commands! With "`extr #2`" in the sequence above, the second move will be affected as well! ATOMIC instructions seem to be a minor issue, since they do not create invalid accesses; in this case the consequence of the bug is that the ATOMIC sequence will be extended to the target instructions also. This is the reason why the compiler does not generate a workaround for ATOMIC

When you use the assembler option **--silicon-bug=12**, the assembler issues an error if it finds a conditional jump inside an extend sequence.

Workaround

Do not jump from extend sequences, or add a NOP at the target location.

CR108219 -- Old value of SP used when second operand of SCXT points to SP

Infineon reference

CR108219

Command line option

`--silicon-bug=13`

Libraries

`lib\p1\*.lib`

Description

A problem can occur when the second operand of SCXT points to SP. In that case the new SP value rather than the old one is written to the first operand.

When you use the assembler option **--silicon-bug=13**, the assembler issues an error if this problem occurs.

CR108309 -- MDL access immediately after a DIV causes wrong PSW values

Infineon reference

CR108309

Command line option

`--silicon-bug=14`

Libraries

`lib\p1\*.lib`

Description

If the MDL register is accessed immediately after a DIV instruction, the PSW flags are set incorrectly. The problem only appears with DIVs instructions when they are immediately followed by one instruction that reads MDL, from type:

```
MOVs mem,reg
ADDs/SUBs/ORs/XORs mem,reg
ADDs/SUBs/ORs/XORs reg,mem
ADDs/SUBs/ORs/XORs reg,mem
CMPs reg.mem
CMPs reg, #data16
```

The V flag can be wrongly calculated for signed divisions: the V flag is only set if the most significant bit of the result is set (that is, if the result is negative).

When you use the assembler option **--silicon-bug=14**, the assembler issues a warning when an instruction after a DIV, DIVL, DIVU or DIVLU instruction accesses MDL.

Workaround

Insert a NOP instruction after DIV instructions:

```
DIVL R0
NOP
MOV R1, MDL
```

CR108342 -- Lost interrupt while executing RETI instruction

Infineon reference

CR108342

Command line option

`--silicon-bug=15`

Libraries

No solution in libraries required.

Description

The bug occurs when two interrupts are trying to get into the CPU while a RETI instruction is being executed. In this case it can happen that one interrupt is lost (the first one, even if it has a higher priority). Furthermore, the program flow after the ISR can be broken. Only the RETI instruction is affected by this bug. This is because this instruction is specially managed. The instruction following the RETI is internally marked as not interruptable. This means that no interrupt will be served by the CPU between the RETI and its following instruction. This bug is the consequence of an error in how this special treatment is implemented in the logic, specifically in the generation of the "not interruptable" indication.

This workaround marks the instruction following the RETI as not interruptable, (emulating what the hardware was supposed to do).

When you use the assembler option **--silicon-bug=15**, the assembler issues a warning when it finds insufficiently protected RETI instructions.

CR108361 -- Incorrect (E)SFR address calculated for RETP as last instruction in extend sequence

Infineon reference

CR108361

Command line option

`--silicon-bug=16`

Libraries

`lib\p1\*.lib`

Description

The address of the operand of a RETP can be calculated wrong when that operand is an SFR or an ESFR and the RETP instruction is the last instruction of an extend sequence.

When you use the assembler option **--silicon-bug=16**, the assembler issues a warning when this problem occurs.

CR108400 -- Broken program flow after not taken JMPR/JMPA instruction

Infineon reference

CR108400

Command line option

`--silicon-bug=17`

Libraries

`lib\p1\*.lib`

Description

This bug can occur in two situations:

- If the instruction sequentially following a conditional JMPR and/or JMPA is the target instruction of another previously executed JB, JNB, JNBS, JMPR or JMPA, the program flow can be corrupted when the JMPR/JMPA is not taken.

- If a not-taken JMPR and/or JMPA is inside a loop or a sequence that is executed more than once (caused by a CALL, RET, JMPI, JMPS or TRAP).

The bug occurs because the instruction sequentially following the not-taken jump is fetched from memory but the "identifier" corresponding to this instruction is taken from the jump cache (since this instruction was previously loaded in the jump cache). As a consequence, both instruction an identifier do not match exactly.

When you use the assembler option **--silicon-bug=17**, the assembler issues a warning when this problem occurs.

CR108904 -- DIV/MUL interrupted by PEC when the previous instruction writes in MDL/MDH

Infineon reference

CR108904

Command line option

`--silicon-bug=18`

Libraries

`lib\p1\*.lib`

Description

If the source pointer of PEC/DPEC/EPEC points to the SFR/ESFR area (PD-bus), the read operation to this SFR/ESFR location is not performed when the PEC/DPEC/EPEC interrupts a DIV/MUL instruction in its first execution cycle AND the DIV/MUL instruction follows an instruction writing into MDL/MDH. In this case, apart from the fact that the read operation is not performed, the value that is written into the destination pointer is always FFFFh (default value of the PD-bus).

The instruction sequences affected, are:

```
MOV mdh, Rw    ; or any instruction that writes in
               ; MDH/MDL (see note) using any addressing mode
               ; (also indirect addressing and bitaddr)

DIV Rw         ; or DIVL/DIVLU/DIVU or MUL/MULU
```

> Writes into ESFR addresses (F00Ch and F00Eh instead of MDH (FE0Ch) and MDL (FE0Eh)) cause the same problem.

When you use the assembler option **--silicon-bug=18**, the assembler issues a warning when it finds an unprotected DIV, DIVL, DIVU, DIVLU, MUL or MULU instruction immediately after MDL or when MDH has been changed.

Workaround

There are two possible workarounds:

- Insert a NOP instruction (or another instruction not writing into MDL/MDH) between an instruction writing into MDL/MDH and a DIV/MUL instruction.

- Do not allow interruption of DIV/MUL by using ATOMIC.

Kfm_BR03 -- Pipeline conflict after CoSTORE

STMicroelectronics reference

Kfm_BR03

Command line option

`--silicon-bug=19`

Libraries

`lib\p1\*.lib`

Description

After a CoSTORE instruction with any destination (E)SFR, the (E)SFR cannot be read.

LONDON.1 -- Breakpoint before JMPI/CAL

Infineon / STMicroelectronics reference

LONDON.1

Command line option

```
--silicon-bug=20
```

Description

JMPI

When the program hits a breakpoint right before a JMPI instruction, the first instruction injected in the pipeline will not be processed by the core. This leads to a deny of all interrupts and OCE injection requests. The problem may also occur when single stepping right before a JMPI instruction.

CALLI

The CALLI instruction is not working properly in some cases if it is followed by an injected interrupt. This results in causing a fault in the stack pointer management.

When you use the assembler option **--silicon-bug=20**, the assembler issues a warning when a CALLI instruction is not protected by an ATOMIC sequence of at least length 2.

LONDON.1751 -- Write to core SFR while DIV[L][U] executes

Infineon / STMicroelectronics reference

LONDON.1751

Command line option

`--silicon-bug=21`

Description

The XC16x / Super10 architectures have a problem writing to a CPU SFR while a DIV[L][U] is in progress in the background. In the following situation:

```
DIVU R12
ADD  R13, R14
...
MOV  MSW, ....  ; will destroy the division
...
MOV  R13,MDH
...
```

When you use the assembler option **--silicon-bug=21**, the assembler issues a warning when this problem occurs.

Workaround

There are different ways to solve this problem:

• Not write to a CPU SFR during the DIV operation.

• Stall the pipeline just before a write operation to a CPU SFR. But because interrupts can write to CPU SFRs as well, the entire DIV operation has to be protected from interrupts (unless it is certain that no interrupt writes to a CPU SFR).

• Do not allow interruption of DIV by using ATOMIC. For example:

```
ATOMIC #2
DIV Rx
MOV Ry, MDL/MDH
```

LONDON RETP -- Problem with RETP on CPU SFRs

Infineon / STMicroelectronics reference

LONDON RETP

Command line option

`--silicon-bug=22`

Description

Some derivatives of the XC16x / Super10 architecture have a problem with RETP on CPU SFRs.

When you use the assembler option **--silicon-bug=22**, the assembler issues a warning whenever RETP is used on one of the CPU SFRs

ST_BUS.1 -- JMPS followed by PEC transfer

STMicroelectronics reference

ST_BUS.1

Command line option

`--silicon-bug=25`

Libraries

lib\p1*.lib

Description

When a JMPS instruction is followed by a PEC transfer, the generated PEC source address is false. This results in an incorrect PEC transfer.

The compiler prevents the JMPS instruction from interfering with the PEC transfers by inserting an `ATOMIC #2` instruction before a JMPS instruction. This bypass option can only be used in combination with the extended instruction set. Further more, all JMPS instructions in the interrupt vector table are replaced by CALLS instructions. The compiler will generate an `ADD SP, #04h` instruction in the interrupt frame to delete the return address generated by the CALLS instruction from the system stack.

When you use the assembler option **--silicon-bug=25**, the assembler issues a warning when this problem occurs.

Workaround

Substitute JMPS by the CALLS instruction with 2 POP instructions at the new program location. You can avoid this problem by disabling interrupts by using the ATOMIC #2 instruction before the JMPS.

CPU_MOVB_JB -- Jump-bit wrongly executed after byte manipulation instructions

Infineon reference

CPU_MOVB_JB

Command line option

--silicon-bug=26

Libraries

```
lib\p1\*.lib
```

Description

When a JB/JNB/JBC/JNBS on a GPR follows an instruction that performs a byte write operation (MOVB, ADDB/ ADDBC, ANDB, XORB, ORB, NEGB, CPLB, SUBB/SUBCB) on the same GPR but on the byte that is not been used by the JB/JNB/JBC/JNBS (i.e. the byte where the bit used by the jump is not located), the program flow gets corrupted. That means, the jump may be wrongly taken (or not taken). A side effect of this bug is that further program branches may also lead to illegal instruction fetches (fetches are performed from wrong addresses).

When you use the assembler option **--silicon-bug=26**, the assembler issues a warning when this problem occurs.

Examples

Example 1:

```
; Assume Rx.0 is 0 (x any GPR 0..7)
MOVB RxH, any_value        ; Any Byte write instruction on RxH
JB Rx.0, jump_address      ; WILL BE WRONGLY TAKEN!
```

Example 2:

```
; Assume Rx.y is 1 (x any GPR 0..7; y any bit except bit0)
MOVB RxL, any_value        ; Any Byte write instruction on
                           ; RxL for y= 8..15 or RxH for y= 1..7
JNB Rx.y, jump_address     ; WILL BE WRONGLY TAKEN!
```

Example 3:

```
; Assume Rx.0 is 0(x any GPR 0..7)
MOVB RxH, any_value        ; Any Byte write instruction on R0H
JNB Rx.0, jump_address     ; WILL NOT BE TAKEN!
```

Example 4:

```
; Assume Rx.y is 1(x any GPR 0..7; y any bit except bit0)
MOVB RxL, any_value        ; Any Byte write instruction on
                           ; RxL for y= 8..15 or RxH for y= 1..7
JB Rx.y, jump_address      ; WILL NOT BE TAKEN!
```

Note that the bug is only visible when the bit used for the jump evaluation is set (i.e."1") for all the bits Rx.15 to Rx.1, or not-set (i.e."0") for Rx.0.

Example 5 (BUG NOT VISIBLE):

```
; Assume Rx.0 is 1 (x any GPR 0..7)
MOVB RxH, any_value        ; Any Byte write instruction on RxH
JB Rx.0, jump_address      ; WILL BE CORRECTLY EXECUTED (TAKEN!)
```

Note that the bug exists also when the byte write operation writes into the GPR using indirect addressing mode or memory addressing mode. This situation includes also the use of PECB/DPECB which destination pointer points into a GPR (i.e. data write is performed on a GPR).

Workaround

1. Include a NOP between any byte write instruction on a GPR and a jump-bit instruction on the same GPR but on a bit belonging to a different byte. This workaround does not cover writing to a GPR through PEC/DPEC.

2. To protect against PEC/DPECB also, include a 'ATOMIC #1' before all jump-bit instructions.

Because the compiler does not expect a GPR modification through a memory or indirect addressing mode, nor through a PEC/DPEC transfer, it uses workaround 1.

Chapter 19. 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.

19.1. 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.

19.2. 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 */
}
```

19.3. 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.

19.4. 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.

19.5. 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.

19.6. 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.

19.7. 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.

19.8. 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.

19.9. 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.

19.10. 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.

19.11. 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 20. MISRA-C Rules

This chapter contains an overview of the supported and unsupported MISRA C rules.

20.1. MISRA-C:1998

This section lists all supported and unsupported MISRA-C:1998 rules.

See also Section 4.8.2, *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.

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.

20.2. MISRA-C:2004

This section lists all supported and unsupported MISRA-C:2004 rules.

See also Section 4.8.2, *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)　`typedef`s that indicate size and signedness should be used in place of the basic types.

6.4　(R)　Bit-fields shall only be defined to be of type `unsigned int` or `signed int`.

6.5　(R)　Bit-fields of type `signed int` shall be at least 2 bits long.

Constants

7.1　(R)　Octal constants (other than zero) and octal escape sequences shall not be used.

Declarations and definitions

8.1　(R)　Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.

8.2　(R)　Whenever an object or function is declared or defined, its type shall be explicitly stated.

8.3　(R)　For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.

8.4　(R)　If objects or functions are declared more than once their types shall be compatible.

8.5　(R)　There shall be no definitions of objects or functions in a header file.

8.6　(R)　Functions shall be declared at file scope.

8.7　(R)　Objects shall be defined at block scope if they are only accessed from within a single function.

8.8　(R)　An external object or function shall be declared in one and only one file.

8.9　(R)　An identifier with external linkage shall have exactly one external definition.

x　8.10　(R)　All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.

8.11　(R)　The `static` storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.

8.12　(R)　When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.

Initialization

9.1   (R)   All automatic variables shall have been assigned a value before being used. This rule is checked using worst-case assumptions. This means that violations are reported not only for variables that are guaranteed to be uninitialized, but also for variables that are uninitialized on some execution paths.

9.2   (R)   Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.

9.3   (R)   In an enumerator list, the "=" construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.

Arithmetic type conversions

10.1   (R)   The value of an expression of integer type shall not be implicitly converted to a different underlying type if:
a) it is not a conversion to a wider integer type of the same signedness, or
b) the expression is complex, or
c) the expression is not constant and is a function argument, or
d) the expression is not constant and is a return expression.

10.2   (R)   The value of an expression of floating type shall not be implicitly converted to a different type if:
a) it is not a conversion to a wider floating type, or
b) the expression is complex, or
c) the expression is a function argument, or
d) the expression is a return expression.

10.3   (R)   The value of a complex expression of integer type may only be cast to a type of the same signedness that is no wider than the underlying type of the expression.

10.4   (R)   The value of a complex expression of floating type may only be cast to a type that is no wider than the underlying type of the expression.

10.5   (R)   If the bitwise operators ~ and << are applied to an operand of underlying type `unsigned char` or `unsigned short`, the result shall be immediately cast to the underlying type of the operand.

10.6   (R)   A "U" suffix shall be applied to all constants of `unsigned` type.

Pointer type conversions

11.1   (R)   Conversions shall not be performed between a pointer to a function and any type other than an integral type.

11.2   (R)   Conversions shall not be performed between a pointer to object and any type other than an integral type, another pointer to object type or a pointer to void.

11.3   (A)   A cast should not be performed between a pointer type and an integral type.

11.4   (A)   A cast should not be performed between a pointer to object type and a different pointer to object type.

11.5   (R)   A cast shall not be performed that removes any `const` or `volatile` qualification from the type addressed by a pointer.

Expressions

12.1	(A)	Limited dependence should be placed on C's operator precedence rules in expressions.
12.2	(R)	The value of an expression shall be the same under any order of evaluation that the standard permits. This rule is checked using worst-case assumptions. This means that a violation will be reported when a possible alias may cause the result of an expression to be evaluation order dependent.
12.3	(R)	The `sizeof` operator shall not be used on expressions that contain side effects.
12.4	(R)	The right-hand operand of a logical `&&` or `||` operator shall not contain side effects.
12.5	(R)	The operands of a logical `&&` or `||` shall be *primary-expressions*.
12.6	(A)	The operands of logical operators (`&&`, `||` and `!`) should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to operators other than (`&&`, `||` and `!`).
12.7	(R)	Bitwise operators shall not be applied to operands whose underlying type is signed.
12.8	(R)	The right-hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left-hand operand. This violation will only be checked when the shift count evaluates to a constant value at compile time.
12.9	(R)	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.
12.10	(R)	The comma operator shall not be used.
12.11	(A)	Evaluation of constant unsigned integer expressions should not lead to wrap-around.
12.12	(R)	The underlying bit representations of floating-point values shall not be used. A violation is reported when a pointer to a floating-point type is converted to a pointer to an integer type.
12.13	(A)	The increment (`++`) and decrement (`--`) operators should not be mixed with other operators in an expression.

Control statement expressions

13.1	(R)	Assignment operators shall not be used in expressions that yield a Boolean value.
13.2	(A)	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.
13.3	(R)	Floating-point expressions shall not be tested for equality or inequality.
13.4	(R)	The controlling expression of a `for` statement shall not contain any objects of floating type.
13.5	(R)	The three expressions of a `for` statement shall be concerned only with loop control. A violation is reported when the loop initialization or loop update expression modifies an object that is not referenced in the loop test.
13.6	(R)	Numeric variables being used within a `for` loop for iteration counting shall not be modified in the body of the loop.
13.7	(R)	Boolean operations whose results are invariant shall not be permitted.

Control flow

14.1   (R)   There shall be no unreachable code.

14.2   (R)   All non-null statements shall either:
a) have at least one side effect however executed, or
b) cause control flow to change.

14.3   (R)   Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment provided that the first character following the null statement is a white-space character.

14.4   (R)   The `goto` statement shall not be used.

14.5   (R)   The `continue` statement shall not be used.

14.6   (R)   For any iteration statement there shall be at most one break statement used for loop termination.

14.7   (R)   A function shall have a single point of exit at the end of the function.

14.8   (R)   The statement forming the body of a `switch, while, do ... while` or `for` statement be a compound statement.

14.9   (R)   An `if (`*expression*`)` construct shall be followed by a compound statement. The `else` keyword shall be followed by either a compound statement, or another `if` statement.

14.10 (R)   All `if ... else if` constructs shall be terminated with an `else` clause.

Switch statements

15.1   (R)   A switch label shall only be used when the most closely-enclosing compound statement is the body of a `switch` statement.

15.2   (R)   An unconditional `break` statement shall terminate every non-empty `switch` clause.

15.3   (R)   The final clause of a switch statement shall be the `default` clause.

15.4   (R)   A `switch` expression shall not represent a value that is effectively Boolean.

15.5   (R)   Every `switch` statement shall have at least one `case` clause.

Functions

16.1   (R)   Functions shall not be defined with variable numbers of arguments.

16.2   (R)   Functions shall not call themselves, either directly or indirectly. A violation will be reported for direct or indirect recursive function calls in the source file being checked. Recursion via functions in other source files, or recursion via function pointers is not detected.

16.3   (R)   Identifiers shall be given for all of the parameters in a function prototype declaration.

16.4   (R)   The identifiers used in the declaration and definition of a function shall be identical.

16.5   (R)   Functions with no parameters shall be declared with parameter type `void`.

16.6   (R)   The number of arguments passed to a function shall match the number of parameters.

16.7   (A)   A pointer parameter in a function prototype should be declared as pointer to `const` if the pointer is not used to modify the addressed object.

16.8   (R)   All exit paths from a function with non-void return type shall have an explicit `return` statement with an expression.

16.9   (R)   A function identifier shall only be used with either a preceding `&`, or with a parenthesized parameter list, which may be empty.

16.10 (R)   If a function returns error information, then that error information shall be tested. A violation is reported when the return value of a function is ignored.

Pointers and arrays

x   17.1   (R)   Pointer arithmetic shall only be applied to pointers that address an array or array element.

x   17.2   (R)   Pointer subtraction shall only be applied to pointers that address elements of the same array.

17.3   (R)   `>`, `>=`, `<`, `<=` shall not be applied to pointer types except where they point to the same array. In general, checking whether two pointers point to the same object is impossible. The compiler will only report a violation for a relational operation with incompatible pointer types.

17.4   (R)   Array indexing shall be the only allowed form of pointer arithmetic.

17.5   (A)   The declaration of objects should contain no more than 2 levels of pointer indirection. A violation is reported when a pointer with three or more levels of indirection is declared.

17.6   (R)   The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

Structures and unions

18.1   (R)   All structure or union types shall be complete at the end of a translation unit.

18.2   (R)   An object shall not be assigned to an overlapping object.

x   18.3   (R)   An area of memory shall not be reused for unrelated purposes.

18.4   (R)   Unions shall not be used.

Preprocessing directives

19.1   (A)   `#include` statements in a file should only be preceded by other preprocessor directives or comments.

19.2   (A)   Non-standard characters should not occur in header file names in `#include` directives.

x   19.3   (R)   The `#include` directive shall be followed by either a `<filename>` or `"filename"` sequence.

19.4   (R)   C macros shall only expand to a braced initializer, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct.

19.5 (R) Macros shall not be #define'd or #undef'd within a block.

19.6 (R) #undef shall not be used.

19.7 (A) A function should be used in preference to a function-like macro.

19.8 (R) A function-like macro shall not be invoked without all of its arguments.

19.9 (R) Arguments to a function-like macro shall not contain tokens that look like preprocessing directives. A violation is reported when the first token of an actual macro argument is '#'.

19.10 (R) In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##.

19.11 (R) All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator.

19.12 (R) There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.

19.13 (A) The # and ## preprocessor operators should not be used.

19.14 (R) The defined preprocessor operator shall only be used in one of the two standard forms.

19.15 (R) Precautions shall be taken in order to prevent the contents of a header file being included twice.

19.16 (R) Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.

19.17 (R) All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.

Standard libraries

20.1 (R) Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined.

20.2 (R) The names of standard library macros, objects and functions shall not be reused.

x   20.3 (R) The validity of values passed to library functions shall be checked.

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.

Chapter 21. Migrating from the Classic Tool Chain to the VX-toolset

The technology used for the tools in the TASKING VX-toolset for C166 is completely different than the technology used for the tools in the classic toolset. The VX-toolset is based on the latest TASKING technology which makes it possible to achieve the significant optimization improvements and additional features such as ISO C99 compliance, MISRA-C 2004, run-time error checking and profiling. Several features have been made more consistent than in the classic toolset.

A drawback of these changes is that both toolsets are not fully compatible, and that existing projects will have to be migrated. The effort needed for such migration depends on the project structure. For example, if a project relies on compiler internals such as a calling convention, work is required to change this code.

This chapter describes how to migrate from the classic C166/ST10 toolset to the TASKING VX-toolset. Also the differences between the C++ compiler, C compiler, assembler and linker in the classic C166/ST10 toolset and the TASKING VX-toolset for C166 are described.

21.1. Importing an EDE Project in Eclipse

Eclipse has the EDE Import Wizard to make importing an existing EDE project easy. This wizard sets up your project, converts EDE option settings as far as possible, invokes the **cnv2vx** tool for source file conversion and generates an LSL file template from an existing ILO file, using the **ilo2lsl** tool:

1. From the **File** menu, select **Import...**

   The Import dialog appears.

2. Select **TASKING C/C++ » TASKING C166 EDE Projects** and click **Next**.

   The Import TASKING C166 EDE Projects dialog appears.

3. Use the **Add...** button to add projects to be converted to the list.

4. Enable options as needed (see description of options below).

5. Click **Finish**.

The import wizard will not modify your original EDE project or your original source files. It copies everything into a new Eclipse project under the current workspace.

Explanation of the options in the import wizard:

Add new C Startup code

Adds a copy of the startup code from the library sources to your project. The startup code settings of the existing project are not imported and need to be set manually afterwards.

Convert C/C++ and assembly source files

When this option is enabled, the C, C++ and assembly source files are converted using the **cnv2vx** tool. A copy of the original files will be saved in the directory `cnv`. For more information on this tool see Section 21.2, *Conversion Tool cnv2vx*.

Convert linker invocation file (.ilo) to template for linker script file (.lsl)

The linker invocation file (`.ilo`) of the classic toolset is generated from the EDE from the Linker/Locator settings. In most cases it is impossible to convert the ILO file into LSL because address ranges for sections must be specified with the correct space in LSL. The ILO file does not provide clues to which space a section, group or class belongs. Also the section names generated by the C compiler are different between both toolsets. Therefore the used section names in the ILO file will not match when you use them in an LSL file. When the option **Convert linker invocation file (.ilo) to template for linker script file (.lsl)** is enabled the **ilo2cnv** tool will be used to convert the ILO file to an LSL template file, with the name *project*`.lsl.template`. The EDE import wizard always adds the default LSL file *project*`.lsl` to your project. You can then update this file with snippets copied from the template file later on.

For more information see Section 21.3, *Conversion Tool ilo2lsl*.

Skip old 'start.asm'

You cannot use the C startup code of the classic toolset with the VX-toolset. When this option is enabled, the file `start.asm` will not be imported in the project.

Import additional files

When this option is enabled files that are not recognized as C, C++ or assembly files can be imported. With the **Additional files filter** you must specify which files to import. By default, these are the C header files (`*.h`) and the assembly include files (`*.inc`). The list of files must be semicolon separated. A file specification can use wild-cards.

Manual actions after import

After importing the project with the wizard, the following manual actions should be performed.

- Check and update where necessary the option settings for your project in the **Project » Properties for » C/C++ Build » Settings** dialog.

- Check the `.log` files in the `cnv` directory that were produced while importing the source code. The entries marked with CHANGED, mark source lines that were changed. More interesting are the lines with WARNING, because these point to lines that could possibly not be converted properly. You will have to inspect those lines in your source code and update them where necessary. For more information about the differences see Section 21.6, *C Compiler Migration* and Section 21.7, *Assembler Migration*.

- In Eclipse the startup code settings can be updated using the startup code editor. The startup code settings are not converted from the classic toolset. You now have two options: update the startup code manually, or use the Import Board Configuration wizard.

The latter is recommended when using a standard evaluation board configuration. The Import Board Configuration Wizard is described in the Getting Started manual. For projects that are built for a custom board or custom configuration of an evaluation board, it is recommended to change the settings manually.

To update the startup code manually, open the Properties dialog to **Project » Properties for » C/C++ Build » Startup Registers**. Update the SFR values. On the **Startup Configuration** property page you can make generic settings for the startup code. For example, for enabling the assignment of the user stack pointer in a local register bank. The size of the allocated stack for local register banks must be specified in the LSL file.

• Once you get the project compiled (and possibly linked), you will have to take a closer look at the LSL file. The template LSL file that resulted from conversion from the ILO file may be of help here, but this requires some knowledge of LSL. You can copy the parts that you want to use from the template LSL file (*project*.lsl.template). See also Section 21.3, *Conversion Tool ilo2lsl*. Alternatively, you can use the **Memory**, **Stack/Heap** and **Vector Table** property pages in the Properties dialog (**Project » Properties for » C/C++ Build**) to make changes to the LSL file. The LSL file *project*.lsl is updated accordingly.

21.2. Conversion Tool cnv2vx

The tool **cnv2vx** converts C, C++ and assembly source files from the classic toolset to the VX-toolset. The tool is located in the cnv2vx directory in the installed product. The invocation syntax is:

cnv2vx [[*option*]... [*file*]...]...

It is possible to add a list of files to **cnv2vx** at once, even with different suffixes. Files with unknown suffixes are not converted.

The converter updates the input file itself. The original file is saved as *file*.org.

You can specify the following options.

Option	Description
-h	Display options
-u	One level undo
-d *directory*	Directory of the search-and-replace tables
-l *file*	Save changes in a log file
-f *file*	Read input from a file

The conversion tool uses search-and-replace tables, which are present in the files *.cnv in the cnv2vx subdirectory of the installed product. The used search-and-replace table file depends on the filename suffix of the source file. For example, a *file*.asm uses the file asm.cnv.

The tool **cnv2vx** is also invoked by the EDE import wizard in Eclipse. See Section 21.1, *Importing an EDE Project in Eclipse*.

21.3. Conversion Tool ilo2lsl

The TASKING VX-toolset for C166 includes the conversion tool **ilo2lsl**. This tool is capable of converting classic linker invocation files (`.ilo`) to the new linker script language files (`.lsl`).

Not all controls given by the invocation file are converted. The converted controls are printed in the output LSL file. The controls that are not converted are printed in the log information.

The C compiler of the VX-toolset for C166 generates different section names than the classic C compiler. The ilo converter is not aware of these generated sections names; the output file will contain the sections names given by the invocation file. Check the output file and change the generated section names with the new generated section names.

The ilo converter does not know the right memory space for sections; the ilo converter generates the string `<fill with proper space value>` instead in the output file. Check the output file for this string and change it to the right space value.

The tool is located in the `ilo2lsl` directory in the installed product. The invocation syntax is:

ilo2lsl [*option*]... [**@***ilofile*]

where, *ilofile* is the original `.ilo` invocation file.

You can specify the following options.

Option	Description
-?	Display options
-V	Display version header only
-l*file*	Save changes in a log file. By default, log information is displayed on your screen.
-o*file*	Specify the name of the LSL output file. By default, the file `default.lsl` is used.

The tool **ilo2lsl** is also invoked by the EDE import wizard in Eclipse. See Section 21.1, *Importing an EDE Project in Eclipse*.

For more information on using LSL see Section 8.7, *Controlling the Linker with a Script*.

21.4. Converting Command Line Options and Makefiles

Most command line options for all tools have changed. The classic assembler and linker used to be controlled with command line controls. The new assembler and linker are controlled using command line options. The locate process is controlled using the LSL language. Due to these differences it is almost impossible to get an automatic and useful conversion of command line invocation. See the comparison tables for the conversion of command line options and controls.

21.5. C++ Compiler Migration

This section describes the migration of the C++ compiler options. The C++ compiler supports the pragmas and language extensions of the C compiler, so for more information on this refer to Section 21.6, *C Compiler Migration*.

The following table shows the options of the classic C166/ST10 C++ Compiler and the possible equivalents in the new TASKING VX-toolset for C166 C++ compiler.

Classic C++ Compiler	VX-toolset C++ Compiler	Remarks
-# --timing	--timing	
-$ --dollar	--dollar	
-?	-? --help	
-A --strict	-A --strict	
-a --strict-warnings	-a --strict-warnings	
--[no-]alternative-tokens	--alternative-tokens	
--[no-]anachronisms	--anachronisms	
--[no-]arg-dep-lookup	--no-arg-dep-lookup	
--[no-]array-new-and-delete	--no-array-new-and-delete	
-b --cfront-2.1		Obsolete
-B --[no-]implicit-include	-B --implicit-include	
--[no-]base-assign-op-is-default	--base-assign-op-is-default	
--[no-]bool	--no-bool	
--[no-]brief-diagnostics		No longer supported
--building-runtime	--building-runtime	
--building-stlport		No longer required to build the STLport library
-C --comments	-Ec --preprocess=+comments	
--cfront-3.0		Obsolete
--[no-]class-name-injection	--no-class-name-injection	
--[no-]const-string-literals	--no-const-string-literals	
--create-pch	--create-pch=*file*	
-D*macro*[(*parm-list*)][=*def*] --define *macro*[(*parm-list*)][=*def*]	-D*macro*[(*parm-list*)][=*def*] --define=*macro*[(*parm-list*)][=*def*]	
--diag-error *tag*...		No longer supported
--diag-remark		No longer supported
--diag-suppress		No longer supported
--diag-warning		No longer supported

Classic C++ Compiler	VX-toolset C++ Compiler	Remarks
--display-error-number		No longer supported
--[no-]distinct-template-signatures	--no-distinct-template-signatures	
-E --preprocess	-E[*flag*],... --preprocess[=*flag*,...]	
-e*num* --error-limit *num*	-e*num* --error-limit=*num*	
--early-tiebreaker		Default. Use --late-tiebreaker to disable.
--[no-]embedded		Obsolete
--embedded-c++	--embedded-c++	
--[no-]enum-overloading	--no-enum-overloading	
--error-output *file*	--error-file[=*file*]	
--[no-]explicit	--no-explicit	
--extended-variadic-macros	--extended-variadic-macros	
-f *file*	-f *file* --option-file=*file*	
--force-vtbl	--force-vtbl	
--[no-]for-init-diff-warning	--no-for-init-diff-warning	
--[no-]friend-injection	--friend-injection	
--[no-]guiding-decls	--guiding-decls	
-H --trace-includes	--trace-includes	
-I*dir* --include-directory *dir*	-I*dir*,... --include-directory=*dir*,...	
--[no-]implicit-extern-c-type-conversion	--implicit-extern-c-type-conversion	
--[no-]implicit-typename	--no-implicit-typename	
--incl-suffixes *suffixes*	--incl-suffixes=*suffixes*	
--include-file *file*	-H*file* --include-file=*file*	
--[no-]inlining	--no-inlining	
--instantiation-dir *dir*		No longer available
-j --no-use-before-set-warnings	-j --no-use-before-set-warnings	
--late-tiebreaker	--late-tiebreaker	
-L*file* --list-file *file*	-L*file* --list-file=*file*	
--long-lifetime-temps	--long-lifetime-temps	
--[no-]long-preserving-rules		No longer available
-M --dependencies	-Em --preprocess=+make	
-M{tsmlh}	-M{nfsh} --model=*model*	The memory model implementation in the new toolset is different
-n --no-code-gen	--check	
--[no-]namespaces	--[no-]namespaces	

Classic C++ Compiler	VX-toolset C++ Compiler	Remarks
--new-for-init		Default. Use --old-for-init to disable.
--[no-]nonconst-ref-anachronism	--nonconst-ref-anachronism	
--[no-]nonstd-qualifier-deduction	--nonstd-qualifier-deduction	
--[no-]nonstd-using-decl	--nonstd-using-decl	
--no-preproc-only	--no-preprocessing-only	
-o*file* --gen-c-file-name *file*	-o*file* --output-file=*file*	
--old-for-init	--old-for-init	
--old-line-commands	--old-line-commands	
--[no-]old-specializations	--old-specializations	
--old-style-preprocessing		No longer available
--one-instantiation-per-object		No longer available
--output *file*	-o*file* --output-file=*file*	Used in combination with -E / --preprocess
-P --no-line-commands	-Ep --preprocess=+noline	
--pch	--pch	
--pch-dir *dir*	--pch-dir=*dir*	
--[no-]pch-messages	--no-pch-messages	
--pch-verbose	--pch-verbose	
--pending-instantiations *n*	--pending-instantiations=*n*	
-r --remarks	-r --remarks	
--[no-]remove-unneeded-entities	--remove-unneeded-entities	
--[no-]rtti	--rtti	
-s --signed-chars	-s --schar	
--short-lifetime-temps		This is the default.
--[no-]special-subscript-cost	--special-subscript-cost	
--suppress-typeinfo-vars		No longer supported
--suppress-vtbl	--suppress-vtbl	
--sys-include *dir*	--sys-include=*dir*,...	
-T --[no-]auto-instantiation	--no-auto-instantiation	
-t*mode* --instantiate *mode*	-t*mode* --instantiate=*mode*	
--[no-]tsw-diagnostics		No longer supported
--[no-]typename	--no-typename	
-U*name* --undefine *name*	-U*name* --undefine=*name*	
-u --unsigned-chars	-u --uchar	Same option name as in C compiler
--use-pch *file*	--use-pch=*file*	

Classic C++ Compiler	VX-toolset C++ Compiler	Remarks
--[no-]using-std	--using-std	
-v -V --version	-v --version	
--variadic-macros	--variadic-macros	
-w --no-warnings	-w --no-warnings	
--[no-]wchar_t-keyword	--wchar_t-keyword	
--[no-]wrap-diagnostics		No longer supported
-x --exceptions/--no-exceptions	-x --exceptions	
-X*file* --xref *file*	-X*file* --xref-file=*file*	

21.6. C Compiler Migration

This section describes the migration of the C compiler options, pragmas, calling convention, memory models, language extensions and preprocessor symbols.

21.6.1. C Compiler Options

The following table shows the options of the classic C166/ST10 Compiler and the possible equivalents in the new TASKING VX-toolset for C166 C compiler.

Classic C Compiler	VX-toolset C Compiler	Remarks
-?	-?	
-A[*flag*...]	-A[*flag*...]	Flags are different
-AC		Character arithmetic cannot be disabled
-AD		Use --no-clear
-AF	-AF	
-AI		Inlining of C library functions is not supported
-AK		New keywords do not need to be disabled because of double underscore
-AL		Identifiers are unique up to 100 characters
-AM	-AD	
-AP	-AP	
-AS		
-AT		
-AU		
-Av	-Ag	
-AW		
-AX	-AX	

Classic C Compiler	VX-toolset C Compiler	Remarks
-B[*flag*...]	--silicon-bug=*number*[,...]	The number of the silicon bugs is listed in Chapter 18, *CPU Problem Bypasses and Checks*.
-D *macro*[=*def*]	-D *macro*[=*def*]	
-E[mcipx]	-E[cmp]	**i** and **x** flags are not supported
-F[*flag*...]	-F	The option does not have flags
-G *groupname*		Near data sections are not grouped anymore
-H	-H	
-I	-I	
-M{tsml}	-M{nfsh} and --near-functions	The memory model implementation in the new toolset is different
-O *flag*...	-O*flag*...	Flags are different
-P[d]	--user-stack[=*flag*]	Flags are different
-R		Changing section attributes is not possible, renaming the sections is possible
-S		Static allocation of automatics is not possible
-T[*size*],*size2*	--near-threshold=*threshold*	
-U	-U	
-V	-V	
-e	-k	Inverted behavior: originally **-e** removed the output file, while **-k** keeps it
-err	--error-file	
-exit	--warnings-as-errors	
-f *file*	-f *file*	
-g[bfls]	-g[acd]	Flags are different
-gso		
-i *scale*		See implementation differences for details
-m		
-misrac *n*,...	--misrac={**all**|*nr*[-*nr*]},...	
-misrac-advisory-warnings	--misrac-advisory-warnings	
-misrac-required-warnings	--misrac-required-warnings	
-n	-n	
-o *file*	-o *file*	
-r[*nr*[, *name*[,**C**]]]		Register banks always have the full size
-s[i]	-s	The **i** flag is not supported
-t		Use the assembler option **-t**
-u	-u	

Classic C Compiler	VX-toolset C Compiler	Remarks
-w[*number*]	-w[*number*]	
-wstrict		Use a selection of warnings with -w
-x[*flags*]	--cpu=*cpu*	
-z *pragma*		Pragmas having a command line equivalent have a long option (--)

21.6.2. Pragmas

The following list shows all pragmas available in the classic C compiler and a description about if and how they are supported in the VX-toolset C compiler.

Classic Compiler Pragma	VX-toolset compiler
alias/noalias	Not available. This pragma is dedicated for the classic compiler technology. Note that there is a pragma alias, which has a different meaning and has arguments. The compiler will complain if arguments are not present, which allows detection of these pragmas in existing code.
asm asm_noflush endasm	Use the `__asm()` method.
autobita *threshold*	This pragma is no longer supported. There is hardly any useful use of this pragma
autobitastruct	This pragma is no longer supported, use command line option: **--bita-struct-threshold** instead. Note that the option **--bita-struct-threshold** does not move volatile and `const` objects to `__bita` memory anymore.
automatic static	No equivalent present.
align	No equivalent present. There is no real use for this pragma in the VX-toolset for C166.
class	The TASKING VX-toolset for C166 C compiler does not use classes. If sections need to be grouped and placed somewhere together at locate time, it is possible to rename the default sections generated by the compiler using the pragma section. With the linker LSL file it is possible to group sections by using wildcards.
combine	Not supported. It makes no sense to change the combine type.
cse	Not supported.
custack nocustack	Not supported. User stack is created at link time.
clear noclear	Is the same: `#pragma clear / noclear`, except that it now also applies to constant data as well.

Classic Compiler Pragma	VX-toolset compiler
default_attributes save_attributes restore_attributes	You cannot change the attributes of a section.
dfap nodfap	DFAP is dedicated to the classic technology.
eramdata iramdata romdata	`#pragma romdata` and `#pragma no_romdata`
fix_byte_write nofix_byte_write	No equivalent present. This pragma was a partial solution for a CPU problem in an old core.
fragment	The sections are always fragmented so that the linker can perform optimizations
global_dead_store_elim no_global_dead_store_elim	No equivalent present
m166include	Use `__asm( ".include 'myfile.inc'");`
macro nomacro	Identical
noframe	Use the function qualifier `__frame()`
preserve_mulip	Use the qualifier `__frame( PSW, MDC, MDH, MDL )` on the interrupt function
public global	No equivalent present because there is no difference between global and public
regdef *number*	No equivalent present. Register banks are always the full width.
reorder noreorder	Use `#pragma optimize k/K` or `+/-schedule`
savemac nosavemac autosavemac	Use the function qualifier `__frame()`
source nosource	Is the same: `#pragma source/nosource`
size speed	Use `#pragma tradeoff`
stringmem	This pragma is replaced by: `#pragma string_literal_memory` *space*. The space arguments are different.

switch_force_table
switch_smart
	These pragmas are replaced by:
	`#pragma linear_switch`
	`#pragma jump_switch`
	`#pragma binary_switch`
	`#pragma smart_switch`

Classic Compiler Pragma	VX-toolset compiler

switch_tabmem_far
switch_tabmem_near
switch_tabmem_default
	These pragmas are replaced by:
	`#pragma constant_memory __near`
	`#pragma constant_memory __shuge`
	`#pragma constant_memory model`
	Note that this pragma affects the allocation of all constants, initializers and switch tables.

volatile_union novolatile_union	No equivalent present

Some pragmas accepted "default" as an argument in the classic C compiler. This argument has been replaced with "model". In the VX-toolset "default" means that the command line state of an option is restored.

21.6.3. Memory Models

Compared to the classic C166/ST10 tool chain, the memory models are organized differently.

Classic Memory Model	VX-toolset Data Model	VX-toolset Function Selection
Tiny -Mt	near -Mn	near --near-functions
Small -Ms (default)	near -Mn (default)	huge (default)
Medium -Mm	far -Mf	near --near-functions
Large -Ml	far -Mf	huge (default)
n/a	shuge -Ms	near --near-functions
n/a	shuge -Ms	huge (default)
n/a	huge -Mh	near --near-functions
Huge -Mh	huge -Mh	huge (default)

In the VX-toolset for C166, all models have a 64 kB near data range. The pages in this near data range can be scattered through the memory. The `_system` and `_xnear` space of the classic toolset do no longer exist and are covered by the `__near` space.

Near is more restrictive than the near in the classic toolset, because objects are limited to 16 kB and must be kept inpage. But it is possible to extend this by removing the page restriction in the LSL file. But this can only be done if you are sure that there are no near to far, shuge or huge casts, or that the near objects that are cast are located in such a way that they follow the restrictions of the target pointer. The same 16 kB page restriction applies to the stack in the near memory model. Care should be taken with far, shuge and huge pointers to objects on the stack because objects may cross page boundaries or segment boundaries and the compiler uses the begin of the stack as page or segment part in a pointer conversion.

The `include.lsl/arch_c166.lsl` file contains the following construction for defining the near space:

```
space near
{
        ...
```

```
#ifdef __CONTIGUOUS_NEAR
                page_size         = 0x10000 [__PAGE_START..0x10000 - __PAGE_END];
#else
                page_size         = 0x4000 [__PAGE_START..0x4000 - __PAGE_END];
#endif
                ...
        }
```

When the `__CONTIGUOUS_NEAR` macro is defined the near page size is set to 64 kB, meaning that objects/sections can be as large as 64 kB and that they can cross a page boundary. The near space is a 64 kB linear addressable space. This also applies to the user stack section. You can define this macro in your project LSL file before the inclusion of the standard LSL files. Conversions from near pointers to a far, huge or shuge pointer, and conversions from pointers to objects on stack to a far, huge or shuge pointer may lead to run-time errors.

> Note: In all models of the VX-toolset for C166 C compiler, the four DPPs are always pointing to near data. For accessing far, shuge and huge data, the compiler uses EXTx instructions.

See the description of the memory models in Section 1.3.2, *Memory Models* for more information.

21.6.4. Calling Convention

The calling convention of the C compiler is completely different compared to that of the classic C compiler. The registers used for stack pointer, return values, passing parameters and local variables are all different. The reason for the new calling convention is that it permits generating the most optimal code. As a result it is impossible to call a C level function that is not compiled with the new C compiler. Linking in existing objects and libraries is not possible because the object file format is different and because of the different calling conventions. Also calling functions compiled with the old C compiler, that reside in existing ROM or flash cannot be called. When your application interfaces to assembly code using the C calling convention, these assembly routines will have to be adapted to the new calling convention.

Some important register usage differences:

Register usage	Classic	VX-toolset
User stack pointer	R0	R15
Parameter registers	R12 - R15	USR0, R2-R5, RL2-RL5 RH2-RH5, R11-R14
Return registers	USR0, R4, R5	USR0, R2-R5, RL2, R14 (buffer pointer)

See Section 1.12.1, *Calling Convention* for a full description of the new calling convention.

21.6.5. Language Implementation Migration

The language extensions in the VX-toolset for C166 C compiler are slightly different from those in the classic C compiler. The most significant difference is the use of a double underscore.

21.6.5.1. migrate.h

The TASKING VX-toolset for C166 includes a header file `migrate.h`. This header file redefines many of the language extensions in the classic C166/ST10 C compiler to the new language extensions. You can include this file from the command line by using the C compiler option **--include-file**. In many cases this makes it possible to share source files between the classic compiler and the new compiler.

Alternatively you can use the source code converter **cnv2vx**. This tool can convert a bit more than covered by `migrate.h`. But after conversion it is impossible to use the same file for both compilers.

21.6.5.2. Qualifiers

The following list shows all C qualifier keywords available in the classic C compiler and a description about if and how they are supported in the VX-toolset for C166 C compiler.

Classic C Compiler	VX-toolset C Compiler
_bank	No equivalent present. Bank switching was introduced for the 8xC166 derivatives because of the limited memory.
_bit	__bit
_bita	__bita
_bitword	__bita volatile unsigned int
_esfrbit _esfr _sfr _sfrbit _xsfr	No equivalent present, see below for notes.
_far	__far
_huge	__huge
_intrinsic	No direct equivalent present, but similar results can be achieved by using inline assembly and inline functions.
_interrupt	__interrupt
_iram	__iram
_near	__near
_noalign	__unaligned
_packed	__packed__ (The syntax of packed structures has changed. Therefore migrate.h will not convert this keyword. See below.)
_shuge	__shuge
_stackparm	No direct equivalent present. The calling convention has changed anyway, existing routines can interface using the register calling convention.
_system	__near
_using	__registerbank
_atbit	__atbit
_inline	inline

Classic C Compiler	VX-toolset C Compiler
_nousm _usm	__nousm __usm
_at	__at
_localbank	__registerbank
_stacksize	No equivalent present because the stack size is specified at link time
_cached	No equivalent present because implementation is changed
_xnear	__near

Packed structures

The syntax of packed structures has changed. The keyword __packed__ is now a symbol attribute (of the tag symbol).

Classic toolset:

```
_packed struct s
{
    char a;
    int b;
} s;
```

VX-toolset:

```
struct __packed__ s
{
    char a;
    int b;
} s;
```

See Section 1.2, *Changing the Alignment: __unaligned and __packed__*.

21.6.5.3. Intrinsic Functions

All intrinsic functions have direct equivalents, but require an additional leading underscore

New intrinsic functions

The following intrinsics have been added:

Prototype	Description
void __sat(void);	Enable MAC saturation; compiler saves MCW
void __nosat(void);	Disable MAC saturation; compiler saves MCW
void __scale(void);	Enable MAC scaling; compiler saves MCW
void __noscale(void);	Disable MAC scaling; compiler saves MCW
void __enwdt(void);	Enable watchdog timer

Prototype	Description
void __CoMACus(unsigned int a, signed int b);	Multiply accumulate: accu += u16 * s16
void __CoMACus_min(unsigned int a, signed int b);	Multiply accumulate: accu -= u16 * s16
void __CoMULus(unsigned int a, signed int b);	Multiply: accu = u16 * s16
void __CoMUL_min(signed int a, signed int b);	Multiply: accu = -(s16 * s16)
void __CoMULu_min(unsigned int a, unsigned int b);	Multiply: accu = -(u16 * u16)
void __CoMULsu_min(signed int a, unsigned int b);	Multiply: accu = -(s16 * u16)
void __CoMULus_min(unsigned int a, signed int b);	Multiply: accu = -(u16 * s16)

Removed intrinsic functions

The following intrinsics have been removed:

Prototype	Description
void __atomic(int);	Disable interrupts for a number instructions

Implementation Differences in Intrinsic Functions

__int166(unsigned char trapno);

When the trap number is 8, and the selected core is super10 or xc16x, then an sbrk instruction will be generated instead of a trap instruction.

__bfld();

The prototype has changed from:

```
void _bfld( unsigned int operand,
            unsigned int mask, unsigned int value);
```

to:

```
void __bfld( volatile unsigned int * operand,
             unsigned short mask, unsigned short value );
```

__getbit() / __putbit( ) / __testclear() / __testset()

Because these intrinsics are now implemented as macros, the compiler will not check the arguments anymore for being a bit or bit-addressable. Instead, the compiler will just generate code for the current type.

21.6.5.4. SFR Definitions

SFRs are usually defined in SFR files, like `reg167.h`. The classic compiler uses `_esfrbit`, `_esfr`, `_sfr`, `_sfrbit` and `_xsfr` keywords for defining the SFRs. The VX-toolset for C166 C compiler just uses #define's of absolute addresses. Internally the compiler knows which address ranges can be accessed using (E)SFR addressing modes, and will generate optimal code for accessing the SFRs. For bits in SFRs the SFR file defines structures for each SFR and defines the bit to a field in this structure. This way no specific SFR language extensions are necessary, and the SFR file contains only 'standard' C code.

As long as you use the SFRs defined in the SFR files included in the product, you do not have to change anything. If you created a new SFR file or just defined SFRs in your code, you have to replace these SFR definitions with #define's as used in the new SFR files.

Example

Classic definition:

```
_sfr BUSCON0
```

VX-toolset definition:

```
#define ADDR_BUSCON0   0xff0c
#define BUSCON0        (*(volatile unsigned int *) ADDR_BUSCON0 )
```

21.6.6. Preprocessor Symbols

The following table lists the differences in the compiler defined preprocessor symbols.

Classic C Compiler	VX-toolset C Compiler
_DOUBLE_FP	__DOUBLE_FP__
_SINGLE_FP	__SINGLE_FP__
_C166	__C166__, does not expand to version number, but to 1, use __VERSION__ to check on the version
_CPUTYPE	__CORE_*core*__
_MODEL	__MODEL__ and __FUNCTIONS__
_USMLIB	__USMLIB__

Example

Classic toolset:

```
#if _CPUTYPE == 0x1662
```

VX-toolset:

```
#if defined(__CORE_xc16x__) || defined(__CORE_super10__)
```

21.6.7. C Compiler Implementation Differences

This section describes the miscellaneous implementation differences between the classic C166/ST10 C compiler and the VX-toolset for C166 C compiler.

Different stdio implementation

When an application relies on the stdio implementation it will possibly not work or even not build anymore. For example:

`_IOSTRG` flag is no longer in `stdio.h`

`_iobuf` is different

You should rewrite the parts that rely on the stdio implementation.

CPU bits defined in SFR file may have conflicts on multi-bit bit-fields

These bit-fields were not defined in the classic C compiler's SFR files. For example PSW fields like ILVL can cause a conflict when you also defines an ILVL.

To get around this, not including the CPU part of the register file is an option here, but may cause troubles if other SFRs or bits in SFRs are used. You can try to not include the CPU part of the register file and add a local register file that only defines those SFRs or SFR fields that are needed by copying them from the SFR file.

Conflicts on SFR or SFR field names as local variable names or as struct/union field names

This conflict arises when

- a struct or union with field name is defined that is also defined in the SFR file

- SFR or SFR field names used as new local variables

Due to the #define method in the SFR file, this will result in fuzzy syntax errors. In the classic C compiler this was not an issue as SFRs got defined with a special type.

Cannot open include file reg*cpu*.h

The `#include <reg`*cpu*`.h>` fails because these register files do no longer exist. The option **--cpu=**_cpu_ now defines the register file. Remove these `#include` lines from your source.

Forward references to enums are not allowed

This will yield an error message "reference to incomplete enum type". For example:

```
enum test foo( void );

enum test { A=1,B=2,C=3 };

enum test foo( void )
```

```
{
 return B;
}
```

This was allowed by the classic C compiler. Make sure the enum is defined before the reference.

Different sizeof() behavior for bit structures

In the classic C compiler the `sizeof()` operator would return the size in bits for bit structures. This has changed, and `sizeof()` always returns the size in bytes. In the VX-toolset use the `__bitsizeof()` operator to obtain the size in bits.

Changed alignment of character arrays

The alignment of character arrays has changed. Arrays of the type char are aligned at a single byte.

With the type qualifier `__unaligned` you can specify to suppress the alignment of objects or structure members. This can be useful to create compact data structures.

```
typedef struct
{
              char fingerPrint[7];
   __unaligned char date[3];
} IDENTITY;
```

With `__packed__` you can make the whole structure packed. For more information see *Section 1.2, Changing the Alignment: __unaligned and __packed__*.

__atbit()

The __atbit() is still supported for backwards compatibility, but has the following implementation differences:

• the bit must be defined `volatile` explicitly. The compiler will issue a warning if the bit is not defined volatile and make the bit volatile.

• the bitword can be any volatile bit-addressable (`__bita`) object. The compiler will issue a warning if the bit-addressable object was not volatile and make it volatile.

• the bit cannot be global. An `extern` on the bit variable will lead to an unresolved external message from the linker .

Example with the classic C compiler:

```
Module 1:
_bitword bitword;
_bit b _atbit( bitword, 3 );

Module 2:
extern _bit b;
```

The header file `migrate.h` contains the following definitions:

```
typedef volatile unsigned short _bitword;
#define _atbit(b,p) __atbit(b,p)
```

When you build the application using this header file you will get a warning from the compiler on the `_atbit()` definition that the bit must be volatile. And linking will result in an unresolved external of "_b". When you use the conversion tool **cnv2vx**, the resulting code will be:

```
Module 1:
volatile __bita unsigned short bitword;
volatile __bit b __atbit( bitword, 3 );

Module 2:
extern __bit b;
```

This will still result in an unresolved external "_b". The extern of the bit should be manually converted to an extern of the bit-addressable word and a new definition of the bit, using an `__atbit()`.

```
Module 1:
volatile __bita unsigned short bitword;
volatile __bit b __atbit( bitword, 3 );

Module 2:
extern volatile __bita unsigned short bitword;
extern volatile __bit b __atbit( bitword, 3);
```

We recommend not to use `__atbit()` at all for accessing bits, but to use one of the methods described in Section 1.3.4, *Accessing Bits*.

_atbit on SFR

With the classic C compiler, the `_atbit()` can also be used on SFRs:

```
_sfrbit pin0 _atbit( P3,0 );

void foo( void )
{
 pin0 = 1;
}
```

This is not possible with the VX-toolset for C166 C compiler because SFRs are defined completely without `_sfr` and `_esfr` keywords, but with `#define`-s. You can convert this by using the bits as defined in the SFR header file. For example:

```
#define pin0 P3_0
```

Scalable interrupt vector table

The VX-toolset C compiler no longer supports the keyword `__cached`, to tell that a part of the interrupt function prologue, including the SCXT instruction must be inlined in the vector table. The linker will only insert a complete interrupt function in the vector table if possible.

Because of this, the C compiler does not need an option to set the scaling of the interrupt vector table. The `__cached` qualifier had to be used when the interrupt vector table had to be bypassed. This could save some latency because a JMPS is saved. However, when the complete ISR is located inside the vector table, the JMPS is already saved. Therefore, this qualifier is now obsolete.

_localbank(0)

The `_localbank()` function qualifier has been replaced by `__registerbank()`. If the *localbank* value is set to zero the VX-toolset C compiler will generate code to select a global bank through the BANK field of the PSW. The classic C compiler did not do this, this should be considered a bug there.

Register bank naming

The name as used in __registbank(*name*) is prefixed with a double underscore "__" in the generated assembly.

-O[uU] command line option

In the classic toolchain this option could be used to save the interrupt frame on the user stack. This option is not supported anymore.

MAC saturation and scaling

Use the `__(no)sat()` / `__(no)scale()` intrinsics, instead of adjusting the MS / MP bit directly. The compiler will save/restore the MCW register automatically in its prologue when the intrinsics are used.

Floating-point byte order

The word order of floating-point values in memory has changed. Floating-point numbers are stored in IEEE-754 format. Single precisions (float) and double precision (double) are stored in memory as shown below.

Address	+0	+1	+2	+3	+4	+5	+6	+7
Single	mmmmmmmm	mmmmmmmm	emmmmmmm	seeeeeee	....	....	....	....
Double	mmmmmmmm	mmmmmmmm	mmmmmmmm	mmmmmmmm	mmmmmmmm	mmmmmmmm	eeeemmmm	seeeeeee

s = sign, e = exponent, m = mantissa, . = not used

Use of pragma m166include from command line

In the classic toolchain a custom assembly include file could be specified on the command line using:

```
c166 x.c -zm166include-\"myfile.inc\"
```

The alternative for the VX-toolset C compiler is to create an include file `header.h` that contains:

```
__asm( ".include 'myfile.inc'" );
```

And include this file from the command line:

```
c166 x.c --include-file=header.h
```

Silicon workaround for CPU.3 (--silicon-bug=3)

The workaround now follows the official description. The old compiler also generated workaround code for:

```
mov  [rm+#data16],rn
movb rbx,[rm+#data16]
movb [rm+#data16],rbx
```

21.7. Assembler Migration

This section describes how to migrate from the classic C166 assembler to the VX-toolset for C166 assembler.

21.7.1. Assembler Concepts

The Task and Flat Interrupt concept are no longer available in the new VX-toolset. In fact it operates like in the Flat Interrupt concept. This is a concept, which is commonly used in almost all toolsets for any architecture. This means that there is no difference between public and global for symbols, sections and groups.

Still it is possible to do incremental linking. It is possible to link the assembler generated objects into larger relocatable objects ('tasks') and then link these together to an absolute file.

21.7.2. Assembler Directives

All directives in the new assembler start with a dot. For compatibility, the directives of the classic assembler are also known without a dot.

Classic C166 Assembler	VX-toolset Assembler
?FILE/?LINE/?SYMB	Uses DWARF debug information
#[line]	`#line`
ASSUME	`.ASSUME`
BIT	`.EQU` or `.SET`
CGROUP, DGROUP	`.CGROUP`, `.DGROUP`
DB, DW, DDW, DBIT	`.DB`, `.DW`, `.DL`, `.DBIT`
DBFILL, DWFILL, DDWFILL	`.DBFILL`, `.DWFILL`, `.DLFILL`
DEFR, DEFA, DEFB, DEFX	Use register definition file to define registers
DS, DSB, DSW, DSDW	`.DS`, `.DSB`, `.DSW`, `.DSL`
DSPTR, DPPTR, DBPTR	`.DSPTR`, `.DPPTR`, `.DBPTR`
END	`.END`
EQU	`.EQU`
EVEN	`.ALIGN 2`
EXTERN, EXTRN	`.EXTERN`

Classic C166 Assembler	VX-toolset Assembler
LABEL	`.LABEL`
LIT	`.DEFINE`
NAME	`.SOURCE`
PECDEF	Reserve PEC pointers in LSL file or with an assembler section
PROC, ENDP	`.PROC`, `.ENDP`
PUBLIC, GLOBAL	`.GLOBAL`
REGDEF, REGBANK, COMREG	`.REGBANK` or define a section with the register bank
SECTION, ENDS	`.SECTION`, `.ENDS`
SET	`.SET`
SSKDEF	Define system stack in LSL file
TYPEDEC	Define label with `.LABEL`

21.7.3. Assembler and Macro Preprocessor Controls

The new assembler does not support controls on the command line. Instead command line options must be used. In the source code only a subset of the controls are supported. Unsupported controls are simply ignored.

Classic C166 Assembler	VX-toolset Assembler
ABSOLUTE / NOABSOLUTE	Not supported
ASMLINEINFO / NOASMLINEINFO	$ASMLINEINFO / $NOASMLINEINFO or --debug-info (-g)
CASE / NOCASE	--case-insensitive (-c) -> Default is now case sensitive!
CHECKUNDEFINED / NOCHECKUNDEFINED	Not supported
CHECK*cpupr* / NOCHECK*cpupr*	$CHECK / $NOCHECK (see Section 21.7.4, *Mapping of CHECKcpupr*) or --silicon-bug -> now uses numbers instead of names
DATE('*date*')	$DATE control
DEBUG / NODEBUG	$DEBUG control or --debug-info (-g)
DEFINE(*name*[,*replacement*])	.DEFINE directive or --define (-D)
EJECT	$EJECT control
ERRORPRINT [(*err-file*)] / NOERRORPRINT	--error-file[=*file*]
EXT* / NOEXT*	--cpu=*cpu* (-C*cpu*) --core=*core*
FLOAT(*float-type*)	Use globals and externs to match float types.
GEN / GENONLY / NOGEN	--list-format=+macro-expansion (-Lx) --list-format=+macro (-Lm)
GSO	Not supported

Classic C166 Assembler	VX-toolset Assembler
HEADER / NOHEADER	Not supported
INCLUDE(*inc-file*)	.INCLUDE directive
LINES / NOLINES	Not supported
LIST / NOLIST	$LIST / $NOLIST controls
LISTALL / NOLISTALL	--list-format (-L)
LOCALS / NOLOCALS	$LOCALS / $NOLOCALS or --emit-locals
MISRAC(*string*)	Supported via sections
MOD166 / NOMOD166	Not supported
MODEL(*modelname*)	Use .GLOBAL / .EXTERN to match models
OBJECT[(*file*)] / NOOBJECT	--output=*file* / --check
OPTIMIZE / NOOPTIMIZE	$OPTIMIZE / $NOOPTIMIZE or --optimize (-O)
PAGELENGTH(*length*)	$PAGELENGTH control
PAGEWIDTH(*width*)	$PAGEWIDTH control
PAGING / NOPAGING	$PAGING / $NOPAGING control
PEC / NOPEC	Not supported
PRINT[(*print-file*)] / NOPRINT	--list-file (-l)
RESTORE / SAVE	$RESTORE / $SAVE controls
RETCHECK / NORETCHECK	$RETCHECK / $NORETCHECK controls
SEGMENTED / NONSEGMENTED	Not supported
STDNAMES(*std-file*)	--cpu=*cpu* (-C*cpu*)
STRICTTASK / NOSTRICTTASK	Not supported
SYMB / NOSYMB	$SYMB / $NOSYMB or --debug-info (-g)
SYMBOLS / NOSYMBOLS	Not supported
TABS(*number*)	$TABS control
TITLE('title')	$TITLE control
TYPE / NOTYPE	Type checking is always done
WARNING(*number*) / NOWARNING(*number*)	$WARNING / $NOWARNING or --no-warnings (-w)
WARNINGASERROR / NOWARNINGASERROR	--warnings-as-errors
XREF / NOXREF	Not supported

21.7.4. Mapping of CHECKcpupr

The following table shows how the CHECK*cpupr* controls of the classic assembler are mapped to the new silicon bug numbers.

Classic C166 Assembler	VX-toolset Assembler
CHECKBUS18 / BUS18	$CHECK(1)
CHECKC166SV1DIV / C166SV1DIV	$CHECK(10)
CHECKC166SV1DIVMDL / C166SV1DIVMDL	$CHECK(14)
CHECKC166SV1DPRAM / C166SV1DPRAM	$CHECK(11)
CHECKC166SV1EXTSEQ / C166SV1EXTSEQ	$CHECK(12)
CHECKC166SV1MULDIVMDLH / C166SV1MULDIVMDLH	$CHECK(18)
CHECKC166SV1PHANTOMINT / C166SV1PHANTOMINT	$CHECK(9)
CHECKC166SV1SCXT / C166SV1SCXT	$CHECK(13)
CHECKCPU3 / CPU3	$CHECK(3)
CHECKCPU16 / CPU16	$CHECK(5)
CHECKCPU1R006 / CPU1R006	$CHECK(2)
CHECKCPU21 / CPU21	$CHECK(7)
CHECKCPUJMPRACACHE / CPUJMPRACACHE	$CHECK(17)
CHECKCPURETIINT / CPURETIINT	$CHECK(15)
CHECKCPURETPEXT / CPURETPEXT	$CHECK(16)
CHECKLONDON1 / LONDON1	$CHECK(20)
CHECKLONDON1751 / LONDON1751	$CHECK(21)
CHECKLONDONRETP / LONDONRETP	$CHECK(22)
CHECKMULDIV / MULDIV	$CHECK(6)
CHECKPECCP / PECCP	$CHECK(8)
CHECKSTBUS1 / STBUS1	$CHECK(25)

21.7.5. Symbol Types and Predefined Symbols

All symbols types of the classic assembler are known by the new assembler.

The new assembler does not support the predefined symbols of the classic assembler (symbols starting with a question mark, such as ?USRSTACK_TOP). Instead different symbols should be used, such as __lc_ub_userstack. These symbols must be declared as an extern explicitly.

List of mappings:

Classic C166 Assembler	VX-toolset Assembler
?USRSTACK_TOP	__lc_ub_user_stack
?USRSTACK_BOTTOM	__lc_ue_user_stack
?USRSTACK0_TOP	__lc_ub_user_stack0
?USRSTACK0_BOTTOM	__lc_ue_user_stack0
?USRSTACK1_TOP	__lc_ub_user_stack1

Classic C166 Assembler	VX-toolset Assembler
?USRSTACK1_BOTTOM	__lc_ue_user_stack1
?USRSTACK2_TOP	__lc_ub_user_stack2
?USRSTACK2_BOTTOM	__lc_ue_user_stack2
?SYSSTACK_TOP	__lc_ub_system_stack
?SYSSTACK_BOTTOM	__lc_ue_system_stack
?C166_INIT_HEAD	no replacement
?C166_BSS_HEAD	no replacement
?C166_NHEAP_TOP	no replacement
?C166_NHEAP_BOTTOM	no replacement
?C166_FHEAP_TOP	no replacement
?C166_FHEAP_BOTTOM	no replacement
?BASE_DPP0	__lc_base_dpp0
?BASE_DPP1	__lc_base_dpp1
?BASE_DPP2	__lc_base_dpp2
?BASE_DPP3	__lc_base_dpp3

21.7.6. Section Directive Attributes

The new `.SECTION` directive has a different implementation. One of the most significant differences is the way ROM and RAM sections are defined.

In the classic assembler a data section becomes a ROM section automatically when it contains initialized data (e.g. defined with a DW directive). In other cases it would become a RAM section. In fact the assembler even does not care, and the linker makes the decision to make a section ROM or RAM. In the VX-toolset for C166 assembler you must specify the ROMDATA attribute when a section must be placed in ROM.

The VX-toolset for C166 assembler also has the INIT/NOINIT and the CLEAR/NOCLEAR attributes. This makes it possible to let the C startup code initialize or clear a section. When a `.DW` directive is used in a data section as with the classic assembler an error will now be issued that a `.DW` directive cannot be used in an uninitialized section. For example:

```
0000                    2 s1      section data
0000                    3         dsw     10
  |   RESERVED
0013
0014                    4         dw      10
  |   RESERVED
0015
as166 E252: ["t.asm" 4] "DW" cannot be encoded in unitialized or cleared section
```

A choice must be made either to place the section in RAM and let the linker generate a ROM copy, by adding the INIT attribute, or to place the section in ROM only by adding the ROMDATA attribute.

The conversion tool cannot convert the section directive in these cases because it cannot look into the section's contents, nor can it look into sections with which it is combined. Therefore you may run into errors as listed above, even after conversion. You will have to add the ROMDATA or INIT attribute manually.

21.7.6.1. Comparison of Section Types

A new set of section types has been introduced, which better match the compiler's memory types.

Classic Section Type	Replacement
CODE	CODE
DATA	FAR
LDAT	NEAR
PDAT	FAR
HDAT	HUGE
SDAT	SHUGE
BIT	BIT

In the classic assembler the DATA type sections were treated differently depending on whether the $MODEL(SMALL) control was set. In the new assembler it is always required to use an explicit section type. So, DATA sections must be changed into NEAR when $MODEL(SMALL) was set.

21.7.6.2. Comparison of Section Align Types

Not all section align types of the classic assembler are supported. The same result can be achieved in a different way.

Align Type	Replacement
BIT	BIT type sections are bit aligned
BYTE	BYTE
WORD	WORD
DWORD	DWORD
PAGE	PAGE
SEGMENT	PAGE
BITADDRESSABLE	Use BITA section
PECADDRESSABLE	Use NEAR sections with the group() attribute to group all PECADDRESSABLE items together and then use LSL to put them in the desired memory range
IRAMADDRESSABLE	Similar solution as PECADDRESSABLE

21.7.6.3. Comparison of Section Combine Types

In the new assembler there is no PUBLIC level. A section is either private or global. Stacks are always defined at locate time, so stack sections are no longer supported. The differences are listed in the following table:

Combine Type	Replacement
Not specified	Same meaning: Sections are never combined
PRIVATE	PRIVATE
PUBLIC / GLOBAL	GLOBAL
COMMON	Use the MAX attribute on the section definition. The MAX attribute has a slightly different meaning. The largest section will be used. The MAX attribute cannot be used on a CODE section.
SYSSTACK	Not supported. The system stack is allocated in the LSL file as a stack section.
USRSTACK	Not supported. The user stack is allocated as a stack section in the LSL file.
GLBUSRSTACK	Not supported. The user stack is allocated as a stack section in the LSL file.
AT *expression*	Use the AT *address* section attribute. In BIT type sections this is the bit address. In all other sections this is a linear address (__huge).

21.7.7. Macro Preprocessor

The new assembler has a different, built-in, macro preprocessor. The following table shows the replacement of the functions and directives.

Function	Replacement
@[*]DEFINE *macro-name*[()] @ENDD	.MACRO .ENDM
@SET(*macro-variable*, *expression*)	.SET
@EVAL(*expression*)	Expressions are evaluated without the need of a specific function.
@IF(*expression*) @ELSE @ENDI	.IF .ELSE .ENDIF
@WHILE(expression) @ENDW	Often an approximation of the @WHILE behavior can be achieved with: .REPEAT *expression* .IF *expression* .BREAK .ENDIF .ENDREP The expression for the .REPEAT should result in a number that is higher than the maximum times the @WHILE could have ran.
@REPEAT(expression) @ENDR	.REPEAT .ENDREP
@BREAK	.BREAK
@EXIT	No direct replacement. Use multiple .BREAK directives.
@ABORT(*exit-value*)	$MESSAGE(F, "*message*")
@LEN(*string*)	@STRLEN(*string*)

Function	Replacement
@SUBSTR(*string*,*expr1*,*expr2*)	@SUBSTR(*string*,*expr1*, *expr2*)
@MATCH(*macro-string*,[*macro-string*,] *string*)	No direct replacement. Use .DEFINE, .SET and .FOR for similar functionality
@EQS(*string*, *string*)	@STRCMP(*string*,*string*) == 0
@NES(*string*, *string*)	@STRCMP(*string*,*string*) != 0
@LTS(*string*, *string*)	@STRCMP(*string*,*string*) < 0
@LES(*string*, *string*)	@STRCMP(*string*,*string*) < 0 || @STRCMP(*string*,*string*) == 0
@GTS(*string*, *string*)	@STRCMP(*string*,*string*) > 0
@GES(*string*, *string*)	@STRCMP(*string*,*string*) > 0 || @STRCMP(*string*,*string*) == 0
@DEFINED([@] *identifier*)	@DEFINED(*macro*)
@IN	No replacement
@OUT(*string*)	$MESSAGE({ I | W | E | F }, *string*)
@""	;

For simple macro preprocessing it is fairly easy to convert the code. For modules that use complex macro preprocessor operations it is recommended to use the classic **m166** for preprocessing and feed the output to the new assembler. The control program, **cc166**, has an option **--m166** to force processing of `.asm` files with the classic **m166** preprocessor. In Eclipse you can find this option on the assembler's preprocessing options page (**Assembler » Preprocessing » Use classic macro preprocessor (m166)**).

21.7.8. Assembler Implementation Differences

This section describes the miscellaneous implementation differences between the classic C166/ST10 assembler and the assembler of the VX-toolset for C166.

Absolute Jump Targets for PC Relative Jumps

The assembler does not support an absolute number as a jump target for a PC relative jump. For example,

```
JNBS R0.0, 16
```

In the classic assembler this jumps to address 16, relative to the section start. In the new assembler this will only work when the section is absolute (specified with the AT attribute).

In the new assembler you should use a label as a reference, for example the procedure label, which is often already at the beginning of the section:

```
secname   .section code
myproc    .proc near
          ...
          JNBS R0.0, myproc + 16
          ...
myproc    .endp
secname   .ends
```

Interrupt Number Symbols

Interrupt number symbols can be created as follows:

```
secname    .section code
myproc     .proc task INTNR = 12
           ...
myproc     .endp
secname    .ends
```

The symbol INTNR will get the INTNO symbol type and can be used for TRAP instructions. The classic assembler encoded INTNR as an interrupt number symbol for the linker/locator. The new assembler encodes the INTNR symbol as a generic global symbol. This can cause name clashes with other symbols, which were previously encoded in a separate name space.

RETV Mnemonic after Unconditional Jumps

The RETV software mnemonic was used in the classic assembler to avoid warnings on missing return statements in procedures. The mnemonic is also supported in the new assembler with the same effect. However, due to the new assembler detecting program flow branching and subsequently warning on unreachable code, you might have to alter the use of the RETV mnemonic in jump-chain situations slightly to avoid a warning:

```
        ...
        JMP _exit
        RETV ; a warning is issued due to this RETV being unreachable
...     .ENDP
```

In this case, the RETV should be positioned before the final unconditional jump. The assembler assesses the RETV as a non-branching instruction (opposed to the other RETx instructions), so it knows that the unconditional jump following it can be reached:

```
        ...
        RETV ; virtual return mnemonic before actual return-implementation
        JMP _exit
...     .ENDP
```

Alternatively, the use of RETV can be replaced by use of the $RETCHECK/$NORETCHECK controls. This will suppress warnings on missing return statements and overflowing program flow.

Symbol redeclared with .extern

The assembler does not support a local symbol redeclared with .extern. The symbol must be made global. For example,

```
code_main    .section code
_main        .proc  far
_func:
             retv
_main        .endp
code_main    .ends
```

978

```
            .extern _func : far
            .end
```

must be changed to

```
code_main   .section code
_main       .proc  far
            .global _func   ;; added
_func:
            retv
_main       .endp
code_main   .ends
            .extern _func : far
            .end
```

21.8. Linker Migration

This section describes how to migrate from the classic linker/locator to the VX-toolset linker.

21.8.1. Linker Controls

The VX-toolset for C166 linker is controlled by an LSL file. The following table shows how functions achieved by controls available in the classic linker/locator can be achieved in the new linker. No replacement is available if the replacement cell is empty.

Classic Control	Replacement in VX-toolset
ADDRESSES SECTIONS ADDRESSES GROUPS ADDRESSES RBANK ADDRESSES LINEAR	Use LSL to locate sections and groups at absolute addresses. Register banks are normal sections. The linear space is supported by mappings in the LSL file, controlled by __BASE_DPPx macros.
ASSIGN	Use LSL symbol assignment
CASE/NOCASE	Command line option --case-insensitive
CHECKCLASSES / NOCHECKCLASSES	
CHECKFIT / NOCHECKFIT	The linker always checks if a symbol fits
CHECKGLOBALS	
CHECKMISMATCH / NOCHECKMISMATCH	Symbol types are not checked by the linker
CLASSES	Use LSL group/select with an address range to locate whole classes. The class name is mangled in the section name.
COMMENTS / NOCOMMENTS	
DATE('date')	
DEBUG / NODEBUG	Command line option --strip-debug (-S)

Classic Control	Replacement in VX-toolset
EXTEND2 EXTEND22 NOEXTEND2 NOEXTEND22 EXTEND2_SEGMENT191	Core differences are coded in the standard LSL files
FIXSTBUS1 / NOFIXSTBUS1	Not needed, as the vector table is defined by a normal section
GENERAL	
GLOBALS / NOGLOBALS	
GLOBALSONLY	
HEADER / NOHEADER	
HEAPSIZE	Specify heap section(s) in LSL file
INTERRUPT	Specify in LSL file
IRAMSIZE	Specify in LSL file, default size is defined in standard LSL files
LIBPATH	Command line option --library-directory (-L)
LINES / NOLINES	Symbols can be stripped using command line option --strip-debug (-S)
LINK / LOCATE	Command line option --link-only
LISTREGISTERS / NOLISTREGISTERS	
LISTSYMBOLS / NOLISTSYMBOLS	Map file contents is controlled with the --map-file-format (-m) option
LOCALS / NOLOCALS	Symbols can be stripped using command line option --strip-debug (-S)
MAP / NOMAP	Command line option --map-file (-M)
MEMORY	Can be specified in LSL
MEMSIZE	Can be specified in LSL
MISRAC	Command line option --misra-c-report
MODPATH	
NAME	
OBJECTCONTROLS	
ORDER	Can be specified in LSL using group/select
OVERLAY	Can be specified in LSL with load and run addresses
PAGELENGTH	
PAGEWIDTH	
PAGING / NOPAGING	
PRINT / NOPRINT	Command line option --map-file (-M)

Classic Control	Replacement in VX-toolset
PRINTCONTROLS	Command line option --map-file-format (-m)
PUBLICS / NOPUBLICS	
PUBLICSONLY	
PUBTOGLB	
PURGE / NOPURGE	
RENAMESYMBOLS	A rename is not supported, but it is possible to make aliases in LSL for global symbols
RESERVE	Memory can be reserved in LSL
RESOLVEDPP / NORESOLVEDPP	Can be achieved by changing the standard LSL file
SECSIZE	
SET	
SETNOSGDPP	Controlled by __BASE_DPPx macros in the standard LSL files
SMARTLINK	Command line options --optimize
STRICTTASK / NOSTRICTTASK	Task concept is not supported
SUMMARY / NOSUMMARY	Command line option --map-file-format=+memory (-mm)
SYMB / NOSYMB	Command line option --strip-debug (-S)
SYMBOLCOLUMNS	
SYMBOLS / NOSYMBOLS	Command line option --strip-debug (-S)
TASK ...INTNO	Specify in the LSL file with vector_table
TITLE	
TO	Command line option --output (-o)
TYPE / NOTYPE	
VECINIT / NOVECINIT	Specify in the LSL file with vector_table
VECSCALE	Specify in the LSL file with vector_table
VECTAB / NOVECTAB	Specify in the LSL file with vector_table
WARNING / NOWARNING	Command line option --no-warnings (-w)
WARNINGASERROR / NOWARNINGASERROR	Command line option --warnings-as-errors

21.8.2. Section, Class and Group Names

The VX-toolset C compiler has a different section generation than the classic C compiler. Section names are different and the VX-toolset C compiler does not generate a class name for a section. Class names could be changed at the C compiler command line or at source level. Also the grouping of sections is different and uses different group names. A locator invocation that relied on these names, cannot be converted directly. A new naming scheme is needed.

In the classic C compiler the section names could not be altered. The used class names could be changed using the `#pragma class` or the **-Rcl** command line option. In the VX-toolset C compiler it is possible to choose any section name for each memory space. See Section 1.15, *Section Naming* on how to rename sections. In the LSL file you can then select on these names. See Section 16.8.2, *Creating and Locating Groups of Sections* for more information on selecting sections.

There is no direct mapping possible between the classic and VX-toolset for this, which means that the conversion tools cannot convert this decently and manual effort is needed to update the source files, options and linker script file.

21.8.3. Object Files

The object files of the classic tool chain and the VX-toolset for C166 have different object formats. The classic tool chain uses the Altium proprietary a.out166 format and the VX-toolset for C166 uses the ELF/DWARF 3 format. The new toolset cannot read object files of the classic tool chain.

The absolute file produced by the classic linker/locater was converted to IEEE-695. CrossView Pro uses this format for debugging. Third party debuggers use either absolute a.out166 or IEEE-695 as object format.

The linker in the VX-toolset for C166 produces absolute ELF/DWARF 3 as debug format. This requires third parties to update their debuggers. Since ELF/DWARF is a widely accepted industry standard, most of these debugger vendors will already support this and only have to update it for C166 support.

Chapter 22. Migrating from Keil

This chapter describes how to migrate from the Keil toolset to the TASKING VX-toolset for C166. Also the differences between the C compiler and assembler from the Keil toolset and the TASKING VX-toolset for C166 are described.

22.1. Importing a Keil Project in Eclipse

Eclipse has the Keil Import Wizard to make importing an existing Keil project easy. This wizard sets up your project, converts Keil/µVison2 option settings as far as possible, invokes the **keil2vx** tool for source file conversion and adds a standard LSL file and the standard C startup code. From the project options the wizard imports the CPU setting, the memory model and the C compiler preprocessor defines. All other options you will have to set manually. Also the C startup code and LSL file have to be setup again.

1. From the **File** menu, select **Import...**

    The Import dialog appears.

2. Select **TASKING C/C++ » Keil C166 Projects** and click **Next**.

    The Import Keil C166 Projects dialog appears.

3. Use the **Add...** button to add projects (with extension `.uv2`) to be converted to the list.

4. Enable options as needed (see description of options below).

5. Click **Finish**.

The import wizard will not modify your original Keil project or your original source files. It copies everything into a new Eclipse project under the current workspace. The original files are copied to `.org` files. Log files of the conversions are placed in a `keil` directory. After import and conversion you need to synchronize the project using the right-mouse button menu -> Refresh.

Explanation of the options in the import wizard:

Add C Startup code

Adds a copy of the startup code from the library sources to your project. The startup code settings of the existing project are not imported and need to be set manually afterwards.

Skip old 'start*.a66'

You cannot use the Keil startup code with the TASKING VX-toolset. When this option is enabled, the files `start*.a66` will not be imported in the project.

Import additional files

When this option is enabled files that are not recognized as C, C++ or assembly files can be imported. With the **Additional files filter** you must specify which files to import. By default, these are the C header

files (`*.h`) and the assembly include files (`*.inc`). The list of files must be semicolon separated. A file specification can use wild-cards.

Manual actions after import

After importing the project with the wizard, the following manual actions should be performed.

• Check and update where necessary the option settings for your project in the **Project » Properties for » C/C++ Build » Settings** dialog.

• Check the `.log` files in the `keil` directory that were produced while importing the source code. The entries marked with `CHANGED`, mark source lines that were changed. More interesting are the lines with `WARNING`, because these point to lines that could possibly not be converted properly. You will have to inspect those lines in your source code and update them where necessary. For more information about the differences see Section 22.4, *C Compiler Migration* and Section 22.5, *Assembler Migration*.

• In Eclipse the startup code settings can be updated using the startup code editor. The startup code settings are not converted from the Keil toolset. You now have two options: update the startup code manually, or use the Import Board Configuration wizard.

  The latter is recommended when using a standard evaluation board configuration. The Import Board Configuration Wizard is described in the Getting Started manual. For projects that are built for a custom board or custom configuration of an evaluation board, it is recommended to change the settings manually.

  To update the startup code manually, open the Properties dialog to **Project » Properties for » C/C++ Build » Startup Registers**. Update the SFR values. On the **Startup Configuration** property page you can make generic settings for the startup code. For example, for enabling the assignment of the user stack pointer in a local register bank. The size of the allocated stack for local register banks must be specified in the LSL file.

• Once you get the project compiled (and possibly linked), you will have to take a closer look at the LSL file. You can make all changes in the LSL file directly or use the **Memory**, **Stack/Heap** and **Vector Table** property pages in the Properties dialog (**Project » Properties for » C/C++ Build**). The LSL file *project*.lsl is updated accordingly.

22.2. Conversion Tool keil2vx

The tool **keil2vx** converts C, C++ and assembly source files from the Keil toolset to the TASKING VX-toolset for C166. The tool is located in the `keil2vx` directory in the installed product. The invocation syntax is:

keil2vx [[*option*]... [*file*]...]...

It is possible to add a list of files to **keil2vx** at once, even with different suffixes. Files with unknown suffixes are not converted.

The converter updates the input file itself. The original file is saved as *file*.org.

You can specify the following options.

Option	Description
-h	Display options
-u	One level undo
-n	Dry run
-d *directory*	Directory of the search-and-replace tables
-l *file*	Save changes in a log file
-f *file*	Read input from a file

The conversion tool uses search-and-replace tables, which are present in the files `*.keil` in the `keil2vx` subdirectory of the installed product. The used search-and-replace table file depends on the filename suffix of the source file. For example, a `file.a66` uses the file `a66.keil`.

The tool **keil2vx** is also invoked by the Keil import wizard in Eclipse. See Section 22.1, *Importing a Keil Project in Eclipse*.

Import restrictions

You should manually update variable name references with the '_' prefix for the Keil assembly blocks after conversion to the `__asm` keyword.

22.3. µVision2 to Eclipse Migration

The following table shows the options of Keil's µVision2 that are converted to Eclipse equivalents in the TASKING VX-toolset for C166. Options that are not mentioned are not converted.

Keil µVision2 Option	TASKING VX-toolset Equivalent	Remarks
Select Device for Target	C/C++ Build » Processor » Processor selection	Processor name until '-' and spaces are removed
Create Executable	project type: TASKING C166 Application	
Create Library	project type: TASKING C166 Library	
Create HEX File	Linker » Output Format » Generate Intel Hex format file	
Debug Information	C/C++ Compiler » Debugging	Generate symbolic debug information
Memory Model: Tiny/Small/Compact/ Medium/Large/HCompact/HLarge	C/C++ Compiler » Memory Model	Default data: near, far, segmented huge, huge Make unqualified functions near
Data Threshold: near, idata, sdata	C/C++ Compiler » Allocation » Threshold for putting data in __near	

Keil µVision2 Option	TASKING VX-toolset Equivalent	Remarks
Data Threshold: bdata	C/C++ Compiler » Allocation » Threshold for putting data in __bita	
Derivative: 167	Assembler » Preprocessing » Defined symbols	__MOD167__=1 is added
Derivative: ext2	Assembler » Preprocessing » Defined symbols	__MODV2__=1 is added
Use On-chip ROM	changes are made to LSL file	IROM(0-....) is added to LSL file with a maximum of 0xE000
External Memory: x=0-5, m=0/1 (RAM/ROM)	changes are made to LSL file	conversion adds memory range to LSL file, for x=0-2 the corresponding DPP register is changed
Code Optimization Emphasis	C/C++ Compiler » Optimization » Trade-off between speed and size	
Treat 'char' as 'unsigned char'	C/C++ Compiler » Language » Treat "char" variables as unsigned	
Double-precision Floating-point	C/C++ Compiler » Language » Treat double as float	
Preprocessor Symbols Define	C/C++ Compiler » Preprocessing » Defined symbols	
Include Paths (C166)	C/C++ Compiler » Include Paths	
MPL Macro Processing	Assembler » Preprocessing » Use classic macro preprocessor (m166)	
Case Sensitive Symbols	Assembler » Symbols » Case insensitive identifiers	
Include Paths (A166)	Assembler » Include Paths	
Use Memory Layout from Target Dialog		use only Target Dialog options
DPPUSE		The following four DPP options are used
DPP Usage: ndata DPP0-3	changes are made to LSL file	- the corresponding DPP register is changed
DPP Usage: nconst DPP0-3	changes are made to LSL file	- the corresponding DPP register is changed
DPP Usage: ndata range	changes are made to LSL file	- a memory range is added to the LSL file

Keil μVision2 Option	TASKING VX-toolset Equivalent	Remarks
DPP Usage: nconst range	changes are made to LSL file	- a memory range is added to the LSL file
Interrupt Vector Table Address	changes are made to LSL file	run_addr of vector_table is updated

22.4. C Compiler Migration

This section describes the migration of the memory models, language extensions, pragmas, preprocessor symbols and intrinsics.

22.4.1. Memory Models

Compared to the Keil toolset, the memory models are organized differently.

Keil Memory Model	VX-toolset Data Model	VX-toolset Function Selection
Tiny (non-segmented)	near	near
Small	near	near
Compact	far	near
Hcompact	huge	near
Medium	near (default)	huge (default)
Large	far	huge (default)
Hlarge	huge	huge (default)

See the description of the memory models in Section 1.3.2, *Memory Models* for more information.

22.4.2. Language Implementation Migration

The language extensions in the VX-toolset for C166 C compiler are different from those in the Keil C compiler.

22.4.2.1. keil.h

The TASKING VX-toolset for C166 includes a header file `keil.h`. This header file redefines many of the language extensions in the Keil C compiler to the VX-toolset language extensions. You can include this file from the command line by using the C compiler option **--include-file**. In many cases this makes it possible to share source files between the Keil compiler and the VX-toolset compiler.

Alternatively you can use the source code converter **keil2vx**. This tool can convert a bit more than covered by `keil.h`. But after conversion it is impossible to use the same file for both compilers.

22.4.2.2. Keywords

In the Keil toolset bits can be declared with the '^' operator:

```
sbit flag0 = flag^0;
sbit bit0 = P2^0;
```

In the VX-toolset you can declare bits as:

```
volatile flag0 __atbit( flag, 0 );
#define bit0 P2_0
```

The following list shows keywords available in the Keil C compiler and a description about if and how they are converted to the VX-toolset for C166 C compiler.

Keil C Compiler	VX-toolset C Compiler
bdata	__bita
bit	__bit
far	__far
huge	__shuge
idata	__iram
interrupt	__interrupt
near	__near
sbit	__bit
sdata	__near
sfr	No equivalent present. See the following section and Section 1.3.5, *Accessing Hardware from C* for information about SFRs in the VX-toolset.
task	No equivalent present.
using	__registerbank
xhuge	__huge

22.4.2.3. SFR Definitions

SFRs are usually defined in SFR files, like `regxc2287.h`. The VX-toolset for C166 C compiler just uses #define's of absolute addresses. Internally the compiler knows which address ranges can be accessed using (E)SFR addressing modes, and will generate optimal code for accessing the SFRs. For bits in SFRs the SFR file defines structures for each SFR and defines the bit to a field in this structure. This way no specific SFR language extensions are necessary, and the SFR file contains only 'standard' C code.

As long as you use the SFRs defined in the SFR files included in the product, you do not have to change anything. If you created a new SFR file or just defined SFRs in your code, you have to replace these SFR definitions with #define's as used in the new SFR files.

Example

VX-toolset definition:

```
#define ADDR_BUSCON0   0xff0c
#define BUSCON0        (*(volatile unsigned int *) ADDR_BUSCON0 )
```

22.4.2.4. Preprocessor Symbols

The following table lists how the Keil compiler defined preprocessor symbols are converted to VX-toolset equivalents.

Keil C Compiler	VX-toolset C Compiler
__C166__	__C166__ (Note that this macro does not expand to the compiler version number. Use __VERSION__ for this.)
__DATE__	__DATE__
__FILE__	__FILE__
__FLOAT64__	__DOUBLE_FP__
__LINE__	__LINE__
__MODEL__	__MODEL__ and __NEAR_FUNCTIONS__ (Note that these macros are expanded differently)
__TIME__	__TIME__
__STDC__	__STDC__

22.4.2.5. Intrinsic Functions

The following table lists the intrinsic functions of the Keil compiler that are converted to equivalents in the VX-toolset.

Keil Intrinsic	VX-toolset Intrinsic
atomic	Not available.
bfld	__bfld
chkdouble	Not available.
chkfloat	Not available.
crol	__rol
cror	__ror
diswdt	__diswdt
einit	__einit
endatomic	Not available.
idle	__idle
irol	__rol
iror	__ror
lrol	Not available.
lror	Not available.
nop	__nop
pop	__asm("POP %0" : "=w" (val));
prior	__prior

Keil Intrinsic	VX-toolset Intrinsic
priord	Not available.
push	__asm("PUSH %0" : : "=w" (val));
pwdrn	__pwrdn
sof	__sof
srvwdt	__srvwdt
testclear	__testclear
testset	__testset
trap	__int166

22.4.2.6. Library Functions

The following table lists the library functions of the Keil compiler that are converted to equivalents in the VX-toolset compiler.

Keil Function	VX-toolset Function
_getkey	Not available.
_tolower	_tolower
_toupper	_toupper
cabs	cabs
init_mempool	Not available.
ungetchar	Not available.
xcalloc	__xhcalloc
xfree	__xhfree
xinit_mempool	Not available.
xmalloc	__xhmalloc
xmemcmp	memcmp (huge memory model only)
xmemcpy	__memcphhb
xmemmove	memmove (huge memory model only)
xmemset	memset (huge memory model only)
xrealloc	__xhrealloc

22.4.3. Pragmas

The following list shows which pragmas of the Keil C compiler are converted to equivalents in the VX-toolset C compiler. For pragmas that are not supported, the pragma is put between comments together with a warning.

Keil Compiler Pragma	VX-toolset Compiler
asm endasm	Not supported. Use the `__asm()` method.
bytealign	Not supported. Use the qualifier `__unaligned`.
[no]init	`#pragma [no]clear`
noframe	Not supported. Use the function qualifier `__frame()`.
optimize	`#pragma tradeoff`
pack	Not supported. Use `__packed__` and `__unaligned`.
renameclass	The TASKING VX-toolset for C166 C compiler does not use classes. If sections need to be grouped and placed somewhere together at locate time, it is possible to rename the default sections generated by the compiler using the pragma `section`. With the linker LSL file it is possible to group sections by using wildcards.
unsigned_char	Not supported. Use C compiler option **--uchar**.
warning	Not supported. Use C compiler option **--no-warnings**.

22.5. Assembler Migration

This section describes how to migrate from the Keil C166 assembler to the VX-toolset for C166 assembler.

22.5.1. Assembler Directives

The following list shows the assembler directives available in the Keil assembler and a description about if and how they are converted to equivalents in the VX-toolset assembler.

Keil Assembler	VX-toolset Assembler
ASSUME	`.ASSUME`
BIT	`.EQU` or `.SET`
CGROUP, DGROUP	`.CGROUP`, `.DGROUP`
DB, DW, DBIT	`.DB`, `.DW`, `.DBIT`
DEFR, DEFA, DEFB	Use register definition file to define registers
DS, DSB, DSW	`.DS`, `.DSB`, `.DSW`
DSPTR, DPPTR, DBPTR	`.DSPTR`, `.DPPTR`, `.DBPTR`
EQU	`.EQU`
END	`.END`
EVEN	`.ALIGN 2`
EXTERN, EXTRN	`.EXTERN`
LABEL	`.LABEL`
LIT	`.DEFINE`
NAME	`.SOURCE`

Keil Assembler	VX-toolset Assembler
ORG	Not supported
PECDEF	Reserve PEC pointers in LSL file or with an assembler section
PUBLIC, GLOBAL	`.GLOBAL`
PROC, ENDP	`.PROC`, `.ENDP`
REGDEF, REGBANK	`.REGBANK` or define a section with the register bank
SECTION, ENDS	`.SECTION`, `.ENDS`
SET	`.SET`
TYPEDEC	Define label with `.LABEL`

22.5.2. Assembler and Macro Preprocessor Controls

The VX-toolset assembler does not support controls on the command line. Instead command line options must be used. In the source code only a subset of the controls are supported. The following Keil controls are supported by the VX-toolset assembler.

Note that the VX-toolset assembler simply ignores unsupported controls.

Supported Keil Assembler Controls
DATE('*date*')
DEBUG / NODEBUG
EJECT
LIST / NOLIST
PAGELENGTH(*length*)
PAGEWIDTH(*width*)
PAGING / NOPAGING
RESTORE / SAVE
TABS(*number*)
TITLE('title')