MB299-024-00-00
Doc. ver.: 1.4

# M16C v3.0

## C Compiler, Assembler, Linker Reference Guide

TASKING
Embedded software development from Altium.

A publication of

Altium BV

Documentation Department



http://www.tasking.com
http://www.altium.com

CONTENTS

# TABLE OF CONTENTS

TASKING

# CONTENTS

CONTENTS

## MANUAL PURPOSE AND STRUCTURE

### *Windows Users*

The documentation explains and describes how to use the M16C toolchain to program an M16C MCU.

You can use the tools either with the graphical Embedded Development Environment (EDE) or from the command line in a command prompt window.

### *Unix Users*

For UNIX the toolchain works the same as it works for the Windows command line.

Directory paths are specified in the Windows way, with back slashes as in `\cm16c\bin`. Simply replace the back slashes by forward slashes for use with UNIX: `/cm16c/bin`.

### *Structure*

The toolchain documentation consists of a User's Guide which includes a Getting Started section and a separate Reference Guide (this manual).

First you need to install the software and make it run under the licence manager FLEXlm. This is described in Chapter 1, *Software Installation and Configuration*, of the *User's Guide*.

After installation you are ready to follow the *Getting Started* in Chapter 2 of the *User's Guide*.

Next, move on with the other chapters in the User's Guide which explain how to use the compiler, assembler, linker and the various utilities.

Once you are familiar with these tools, you can use the Reference Guide to lookup specific options and details to make fully use of the M16C toolchain.

## SHORT TABLE OF CONTENTS

***Chapter 6: Object File Formats***

Contains a description of the following object file formats:

- ELF/DWARF Object Formats
- Motorola S–Record Format
- Intel Hex Record Format

***Chapter 7: Linker Script Language***

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

***Chapter 8: MISRA C Rules***

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

## CONVENTIONS USED IN THIS MANUAL

### *Notation for syntax*

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

| | |
|---|---|
| **bold** | Type this part of the syntax literally. |
| *italics* | Substitute the italic word by an instance. For example:<br>*filename*<br>Type the name of a file in place of the word *filename*. |
| { } | Encloses a list from which you must choose an item. |
| [ ] | Encloses items that are optional. For example<br>**cm16c** [ **-?** ]<br>Both **cm16c** and **cm16c -?** are valid commands. |
| \| | Separates items in a list. Read it as OR. |
| ... | You can repeat the preceding item zero or more times. |
| ,... | You can repeat the preceding item zero or more times, separating each item with a comma. |

### *Example*

**cm16c** [*option*]... *filename*

You can read this line as follows: enter the command **cm16c** with or without an option, follow this by zero or more options and specify a *filename*. The following input lines are all valid:

```
cm16c test.c
cm16c -g test.c
cm16c -g -E test.c
```

Not valid is:

```
cm16c -g
```

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

### *Icons*

The following illustrations are used in this manual:

Note: notes give you extra information.

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

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

Command line: type your input on the command line.

Reference: follow this reference to find related topics.

## RELATED PUBLICATIONS

***C Standards***

- C A Reference Manual (fifth edition) by Samual P. Harbison and Guy L. Steele Jr. (2002, Prentice Hall)
- The C Programming Language (second edition) by B. Kernighan and D. Ritchie (1988, Prentice Hall)
- ISO/IEC 9899:1999(E), Programming languages – C [ISO/IEC]
  More information on the standards can be found at
  `http://www.ansi.org`
- DSP–C, An Extension to ISO/IEC 9899:1999(E),
  Programming languages – C [TASKING, TK0071–14]

***MISRA C***

- Guidelines for the Use of the C Language in Vehicle Based Software [MISRA]
  See also `http://www.misra.org.uk`

***TASKING Tools***

- M16C C Compiler, Assembler, Linker User's Guide
  [TASKING, MA299–024–00–00]
- M16C C++ Compiler User's Guide
  [TASKING, MA299–012–00–00]
- M16C CrossView Pro Debugger User's Guide
  [TASKING, MA299–041–00–00]

***M16C***

- M16C Group Specification [Renesas]
- M16C/60/20 Series Software Manual [Renesas]

CHAPTER 1

# C LANGUAGE

TASKING

# 1 CHAPTER

## 1.1 INTRODUCTION

The TASKING M16C C compiler fully supports the ANSI C standard but adds possibilities to program the special functions of the M16C.

This chapter contains complete overviews of the following C language extensions of the TASKING M16C C compiler:

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

## 1.2 DATA TYPES

The TASKING C compiler for the M16C architecture supports the following data types:

| Type | Keyword | Size (bit) | Align (bit) | Ranges |
|---|---|---|---|---|
| Bit | `__bit` | 1 | 1 | 0 or 1 |
| Boolean | `_Bool` | 1 | 8 | 0 or 1 |
| Character | `char`<br>`signed char` | 8 | 8 | $-2^7$ .. $2^7-1$ |
| | `unsigned char` | 8 | 8 | 0 .. $2^8-1$ |
| Integral | `short`<br>`signed short`<br>`int`<br>`signed int` | 16 | 8 / 16* | $-2^{15}$ .. $2^{15}-1$ |
| | `unsigned short`<br>`unsigned int` | 16 | 8 / 16* | 0 .. $2^{16}-1$ |
| | `enum` | 1<br>8<br>16 | 8<br>8 / 16*<br>8 / 16* | 0 or 1<br>$-2^7$ .. $2^7-1$<br>$-2^{15}$ .. $2^{15}-1$ |
| | `long`<br>`signed long` | 32 | 8 / 16* | $-2^{31}$ .. $2^{31}-1$ |
| | `long long`<br>`signed long long` | 64 | 8 / 16* | $-2^{63}$ .. $-2^{63}-1$ |
| | `unsigned long` | 32 | 8 / 16* | 0 .. $2^{32}-1$ |
| | `unsigned long long` | 64 | 8 / 16* | 0 .. $2^{64}-1$ |
| Pointer | `pointer to __sfr, __bita` | 16 | 8 / 16* | 0 .. $2^{13}-1$ |
| | `pointer to __near` | 16 | 8 / 16* | 0 .. $2^{16}-1$ |
| | `pointer to __far, __paged` | 32 | 8 / 16* | 0 .. $2^{20}-1$ |
| Floating Point | `float` | 32 | 8 / 16* | $-3.402e^{38}$ .. $-1.175e^{-38}$<br>$1.175e^{-38}$ .. $3.402^{e38}$ |

| Type | Keyword | Size (bit) | Align (bit) | Ranges |
|---|---|---|---|---|
| | `double`<br>`long double` | 64 | 8 / 16* | $-1.797e^{308}$ .. $-2.225e^{-308}$<br>$2.225e^{-308}$ .. $1.797e^{308}$ |
| | `float`<br>`_Imaginary` | 32 | 8 / 16* | $-3.402e^{38}i$ .. $-1.175e^{-38}i$<br>$1.175e^{-38}i$ .. $3.402e^{38}i$ |
| | `float _Complex` | 32+32 | 8 / 16* | real part + imaginary part |
| | `double/`<br>`long double`<br>`_Imaginary` | 64 | 8 / 16* | $-1.797e^{308}i$ .. $-2.225e^{-308}i$<br>$2.225e^{-308}i$ .. $1.797e^{308}i$ |
| | `double/`<br>`long double`<br>`_Complex` | 64+64 | 8 / 16* | real part + imaginary part |

*Table 1–1: Data Types*

* For the marked data types, the alignment is 16 if you specify compiler option **--align**, otherwise the alignment is 8.

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

`float` is implemented in little endian IEEE 32–bit single precision format. `double` is implemented in little endian IEEE 64–bit double precision format.

When you compile for the R8C/tiny (compiler option **--r8c**) `__far` and `__paged` are the same as `__near`.

## 1.3 KEYWORDS

### *__asm()*

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

```
__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* [***.regnum***] |
| **%***parm_nr*[***.regnum***] | Parameter number in the range 0 .. 31. With the optional ***.regnum*** you can access an individual register from a register pair. For example, with the word register `R2R0`, `.0` selects register `R0`. |
| *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. |

| Constraint character | Type | Operand | Remark |
|---|---|---|---|
| a | address register | A0, A1 | word register |
| A | address register | A1A0 | double–word register |
| b | bit | R[0..3]H.[0..7]<br>R[0..3]L.[0..7]<br>A[0..1].[0..7]<br>C<br>*_bitvar* | bit registers/variables |
| h | data register | R[0..3]H<br>R[0..3]L | byte registers |
| i | immediate value | *#value* | |
| m | memory | *address*, *label*, *_variable* | memory variable or function address |
| r | data register | R[0..3] | word registers |
| R | registers | R2R0, R3R1 | double–word registers |
| *number* | other operand | same as **%***number* | used when input and output operands must be the same |

*Table 1–2: Available input/output operand constraints*

Section 3.6, *Using Assembly in the C Source*, in Chapter *C Language* of the *User's Guide*.

### *__at()*

With the attribute `__at()` you can place an object at an absolute address.

```
int myvar __at(0x100);
```

Section 3.4.3, *Declare a Data Object at an Absolute Address*, in Chapter *C Language* of the *User's Guide*.

### *__bita*

With the `__bita` memory type qualifier, you can specify that a variable must be in bitaddressable RAM.

```
__bita  int   array[10][4];
```

Section 3.4.1, *Memory Type Qualifiers*, in Chapter *C Language* of the *User's Guide*.

### *__near, __far, __paged*

With the `__near` memory type qualifier, you can specify that a variable must be placed in the first 64 kB of memory. With `__far` data can be located anywhere in memory. With `__paged` data is located in a 64 kB page, anywhere in memory.

```
__far   int i;
```

Section 3.4.1, *Memory Type Qualifiers*, in Chapter *C Language* of the *User's Guide*.

### *__rom*

With the `__rom` memory type qualifier, you can specify that a variable must be placed in ROM.

```
__rom   char  text[] = "No smoking";
```

Section 3.4.1, *Memory Type Qualifiers*, in Chapter *C Language* of the *User's Guide*.

### *__sfr*

With the `__sfr` memory type qualifier you can define a symbol as a Special Function Register (SFR).

```
#define P0      (*(__sfr unsigned char *)0x00E0)
```

Section 3.4.2, *Accessing Peripherals from C: __sfr*, in Chapter *C Language* of the *User's Guide*.

## 1.4 FUNCTION QUALIFIERS

***__asmfunc***

You can use the `__asmfunc` qualifier for a prototype of an assembly function to be called from C or for a function definition of a C function to be called from assembly. Normally, the C compiler adds a leading underscore when it generates an assembly function, with `__asmfunc` the C compiler does not add the extra underscore.

```
/* prototype of assembly function */
extern __asmfunc int
special_out( int port, long config, int value );

void main( void )
{ ...
/* call assembly function */
     long cfg;
     int y = special_out( 1, cfg, y );
  ...
}
```

***inline***
***__noinline***

You can use the `inline` qualifier to tell the compiler to inline the function body instead of calling the function. Use the `__noinline` qualifier to tell the compiler *not* to inline the function body.

```
inline int func1( void )
{
     // inline this function
}

__noinline int func2( void )
{
     // do not inline this function
}
```

For more information see section 3.12.3, *Inlining Functions: inline*, in Chapter *C Language* of the *User's Guide*.

***__interrupt()***
***__interrupt_fixed()***
***__bankswitch***
***__frame()***

With the following function qualifiers you can declare an interrupt handler using the relocatable or fixed vector table respectively.

```
void __interrupt( vector,... ) isr(void)
{
...
}

void __interrupt_fixed( vector,... ) isr(void)
{
...
}
```

The argument *vector* identifies the interrupt number entry in the interrupt vector table. This number must be in the range 0 to 63 for `__interrupt()` or 0 to 8 for `__interrupt_fixed()`.

With the function qualifier `__bankswitch` you can specify to use register bank 1 for the interrupt function.

```
__interrupt( vector,... ) __bankswitch
void isr( void )
{
...
}
```

With the function qualifier `__frame()` you can specify which registers must be saved for a particular interrupt function. Only the specified registers will be pushed and popped from the stack. The syntax is:

```
__interrupt( vector,... ) __frame( reg,... )
void isr( void )
{
...
}
```

For more information see section 3.12.6, *Interrupt Functions*, in Chapter *C Language* of the *User's Guide*.

## 1.5 INTRINSIC FUNCTIONS

The TASKING M16C C compiler recognizes intrinsic functions that serve the following purposes:

- Arithmetic functions
- Interrupt handling
- Control Register handling
- Block functions
- Bit Data functions
- Miscellaneous functions

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

### 1.5.1 ARITHMETIC FUNCTIONS

The next table provides an overview of the intrinsic functions to perform several arithmetic operations.

| Intrinsic Function | Description |
|---|---|
| `signed char __absb (signed char)` | Return absolute value (8 bit) |
| `int __absw (int)` | Return absolute value (16 bit) |
| `char __dadcb (char, char)` | Decimal add with carry (8 bit) |
| `int __dadcw (int, int)` | Decimal add with carry (16 bit) |
| `char __daddb (char, char)` | Decimal add without carry (8 bit) |
| `int __daddw (int, int)` | Decimal add without carry (16 bit) |
| `int __divb (int, char)` | Returns quotient and remainder |
| `char __divb_q (int, char)` | Returns quotient |
| `char __divb_r (int, char)` | Returns remainder |
| `long int __divw (long int, int)` | Returns quotient and remainder |
| `int __divw_q (long int, int)` | Returns quotient |
| `int __divw_r (long int, int)` | Returns remainder |
| `int __divub (int, char)` | Returns quotient and remainder |
| `char __divub_q (int, char)` | Returns quotient |
| `char __divub_r (int, char)` | Returns remainder |
| `long int __divuw (long int, int)` | Returns quotient and remainder |
| `int __divuw_q (long int, int)` | Returns quotient |
| `int __divuw_r (long int, int)` | Returns remainder |
| `int __divxb (int, char)` | Returns quotient and remainder |
| `char __divxb_q (int, char)` | Returns quotient |
| `char __divxb_r (int, char)` | Returns remainder |
| `long int __divxw (long int, int)` | Returns quotient and remainder |
| `int __divxw_q (long int, int)` | Returns quotient |
| `int __divxw_r (long int, int)` | Returns remainder |
| `char __dsbbb (char, char)` | Decimal subtract with borrow (8 bit) |
| `int __dsbbw (int, int)` | Decimal subtract with borrow (16 bit) |

| Intrinsic Function | Description |
|---|---|
| `char __dsubb (char, char)` | Decimal subtract without borrow (8 bit) |
| `int __dsubw (int, int)` | Decimal subtract without borrow (16 bit) |
| `char __rotb (signed char, char)` | Rotote (8 bit). Signed char specifies direction. |
| `int __rotw (signed char, int)` | Rotote (16 bit). Signed char specifies direction. |
| `char __shab (signed char, char)` | Shift arithtmetic (8 bit) |
| `int __shaw (signed char, int)` | Shift arithmetic (16 bit) |
| `long int __shal (signed char, long int)` | Shift arithmetic (32 bit) |
| `char __shlb (signed char, char)` | Shift logical (8 bit) |
| `int __shlw (signed char, int)` | Shift logical (16 bit) |
| `long int __shll (signed char, long int)` | Shift logical (32 bit) |

*Table 1–3: Intrinsic Functions for Arithmetic Operations*

### 1.5.2 INTERRUPT HANDLING

The next table provides an overview of the intrinsic functions to generate interrupts.

| Intrinsic Function | Description |
|---|---|
| `void __brk (void)` | break interrupt |
| `void __int (int)` | software interrupt (vector number) |
| `void __into (void)` | overflow interrupt |
| `void __und (void)` | interrupt |
| `void __wait (void)` | interrupt |

*Table 1–4: Intrinsic Functions for Interrupt Handling*

### 1.5.3 CONTROL REGISTER HANDLING

***Access Control Registers***

The next table provides an overview of the intrinsic functions that you can use to acces control registers.

| Intrinsic Function | Description |
|---|---|
| `int __fclr (int)` | Use 0 to 7 or __C, __D, __Z, __S, __B, __O, __I, __U to clear a bit in the flag register. |
| `int __fset (int)` | Use 0 to 7 or __C, __D, __Z, __S, __B, __O, __I, __U to set a bit in the flag register. |
| `void __ldctx (_near int, _far long int)` | Restore context |
| `int __ldc_fb (int)` | Load control register fb |
| `int __ldc_sb (int)` | Load control register sb |
| `int __ldc_sp (int)` | Load control register sp |
| `int __ldc_isp (int)` | Load control register isp |
| `int __ldc_flg (int)` | Load control register flg |
| `int __ldc_intbh (int)` | Load control register intb (high) |
| `int __ldc_intbl (int)` | Load control register intb (low) |
| `void __ldintb (_far void *)` | Load control register intb |
| `void __stctx (_near int, _far long int)` | Store context |
| `int __stc_fb (void)` | Store control register fb |
| `int __stc_sb (void)` | Store control register sb |
| `int __stc_sp (void)` | Store control register sp |
| `int __stc_isp (void)` | Store control register isp |
| `int __stc_flg void()` | Store control register flg |
| `int __stc_intbh (void)` | Store control register intb (high) |
| `int __stc_intbl (void)` | Store control register intb (low) |

*Table 1–5: Intrinsic Functions for Accessing Control Registers*

### 1.5.4 BLOCK FUNCTIONS

The next table provides an overview of the intrinsic functions to handle blocks of data.

| Intrinsic Function | Description |
|---|---|
| `int __rmpab (_near char *source, _near char *dest, int count)` | Repeat multiply and addition (8 bit) |
| `long int __rmpaw (_near char *source, _near char *dest, int count)` | Repeat multiply and addition (16 bit) |
| `void __smovbb (_far char *source, _near char *dest, int count)` | String move backward (8 bit) |
| `void __smovbw (_far int *source, _near int *dest, int count)` | String move backward (16 bit) |
| `void __smovfb (_far char *source, _near char *dest, int count)` | String move forward (8 bit) |
| `void __smovfw (_far int *source, _near int *dest, int count)` | String move forward (16 bit) |
| `void __sstrb (char, _near char *, int)` | Store string (8 bit) |
| `void __sstrw (int, _near int *, int)` | Store string (16 bit) |

*Table 1–6: Intrinsic Functions to Handle Blocks of Data*

### 1.5.5 BIT DATA FUNCTIONS

The next table shows intrinsic functions to handle bit data.

| Intrinsic Function | Description |
|---|---|
| `__bit __btstc (__bit *)` | Bit test and clear |
| `__bit __btsts (__bit *)` | Bit test and set |

*Table 1–7: Intrinsic Functions to Handle Bit Data*

### 1.5.6 MISCELLANEOUS INTRINSIC FUNCTIONS

| Intrinsic Function | Description |
|---|---|
| `int __enter (int)` | Build stack frame |
| `void __exitd (void)` | Deallocate stack frame |
| `int __ldipl (char)` | Load interrupt permission level |
| `void __nop (void)` | Insert nop instruction |
| `int __popc (int)` | The operand is the register as encoded in the opcode. |
| `int __popm (int)` | The operand is the register mask as encoded in the opcode. |
| `int __pushc (int)` | The operand is the register as encoded in the opcode. |
| `int __pushm (int)` | The operand is the register mask as encoded in the opcode. |
| `void __reit (void)` | Return from interrupt |
| `void __rts (void)` | Return from subroutine |

*Table 1–8: Miscellaneous Intrinsic Functions*

## 1.6 PRAGMAS

*Pragmas* are keywords in the C source that control the behavior of the compiler. Pragmas overrule compiler options and keywords. The syntax is:

**#pragma** *pragma-spec* [**ON** | **OFF** | **DEFAULT**]

or:

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

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

***#pragma alias*** *symbol=defined_symbol*

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

See the **EQU** directive directive in Section 3.3, *Assembler Directives and Controls*, in Chapter *Assembly Language*.

***#pragma align***
***#pragma align-data***
***#pragma align-func***

By default the compiler aligns objects to the minimum alignment required by the architecture.

With these pragmas you can align objects to even adresses. Pragma `align` aligns all objects to even addreses. Pragma `align-data` aligns all data to even addresses. Pragma `align-func` aligns all functions to even addresses.

See compiler option **--align** in section 4.1, *Compiler Options,* in Chapter *Tool Options*.

***#pragma extension isuffix***

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

***#pragma 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 force the compiler to generate the `.EXTERN` directive, creating an external symbol in the generated assembly source, even when the symbol is not used in the C module.

See the **EXTERN** directive directive in Section 3.3, *Assembler Directives and Controls*, in Chapter *Assembly Language*.

***#pragma clear***
***#pragma 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 behaviour.

See compiler option **–-noclear** in section 4.1, *Compiler Options,* in Chapter *Tool Options*.

***#pragma inline***
***#pragma noinline***

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.

***#pragma smartinline***

By default, small fuctions that are not too often called, are inlined. This reduces execution time at the cost of code size (compiler option **–Oi**).

With the `pragma noinline / pragma smartinline` you can temporarily disable this optimization.

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

See for more information of these options, section 4.1, *Compiler Options* in Chapter *Tool Options*.

***#pragma linear_switch***
***#pragma jump_switch***
***#pragma binary_switch***
***#pragma auto_switch***

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

| | |
|---|---|
| `linear_switch` | force jump chain code |
| `jump_switch` | force jump table code |
| `lookup_switch` | force lookup table code |
| `auto_switch` | let the compiler decide the switch method used |

See Section 3.11, *Switch Statement* in Chapter *C Language* of the *User's Guide*.

***#pragma macro***
***#pragma nomacro***

Turns macro expansion on or off.

***#pragma message "string" ...***

Print the message string(s) on standard output.

***#pragma optimize*** *flags*
***#pragma endoptimize***

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

See compiler option **–O** in section 4.1, *Compiler Options,* in Chapter *Tool Options*.

***#pragma renamesect*** *spec*
***#pragma endrenamesect***

Rename sections of the specified type or restore default section naming.

See section 3.13, *Section Naming* in Chapter *C Language* of the *User's Guide*.
See compiler option **–R** in section *Compiler Options* in Chapter *Tool Options*.

***#pragma source***
***#pragma nosource***

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

See also compiler option **–s (––source)**

***#pragma tradeoff*** *level*

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

See also compiler option **–t (––tradeoff)**

***#pragma warning*** *[number,...]*

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

See also compiler option **–w (––no–warnings)**

***#pragma weak*** *symbol*

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

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

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

See the **WEAK** directive directive in Section 3.3, *Assembler Directives and Controls*, in Chapter *Assembly Language*.

## 1.7 PREDEFINED MACROS

In addition to the predefined macros required by the ISO C standard, 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 |
|---|---|
| __SINGLE_FP__ | Defined when you use compiler option **–F** (Treat double as float) |
| __CM16C__ | Identifies the compiler. You can use this symbol to flag parts of the source which must be recognized by the **cm16c** compiler only. It expands to the version number of the compiler. |
| __CPU__ | Expands to the CPU type specified to the compiler option **–C**, or 0 otherwise. |
| __LITTLE_ENDIAN__ | Expands to 1, indicating the processor accesses data in little–endian. |
| __MODEL__ | Identifies the memory model for which the current module is compiled. For example, if you compile for the small memory model, the macro expands to s. |
| __M16C__ | Defined when you select a M16C core. |
| __R8C__ | Defined when you select a R8C core (**––r8c**). |
| __TASKING__ | Identifies the compiler as a TASKING compiler. It expands to 1. |
| __DSPC__ | Indicates conformation to the DSP–C standard. Expands to 0, DSP–C extensions are not supported. |
| __VERSION__ | Identifies the version number of the compiler. For example, if you use version 3.0r1 of the compiler, __VERSION__ expands to 3000 (dot and revision number are omitted, minor version number in 3 digits). |
| __REVISION__ | Identifies the revision number of the compiler. For example, if you use version 3.0r1 of the compiler, __REVISION__ expands to 1. |
| __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. |

*Table 1–9: Predefined macros*

CHAPTER 2

# LIBRARIES

TASKING

# 2 CHAPTER

## 2.1 INTRODUCTION

This chapter contains an overview of all library functions that you can call in your C source. This includes all functions of the standard C library (`libc`*m*`.a`) and some functions of the floating–point library (`libfp`*m*`.a` or `libfp`*m*`t.a`), where *m* is the model s (small), m (medium) or l (large). The libraries follow the ISO/IEC 9899 standard.

Section 2.2, *Library Functions*, gives an overview of all library functions you can use, grouped per header file. A number of functions declared in `wchar.h` are parallel to functions in other header files. These are discussed together.

The following libraries are included in the M16C (**cm16c**) toolchain. Both EDE and the control program **ccm16c** automatically select the appropriate libraries depending on the specified M16C or R8C derivative.

| Library to link | Description |
|---|---|
| libcs.a<br>libcm.a<br>libcl.a | C library for small, medium or large memory model<br>(Some functions require the floating–point library. Also includes the startup code.) |
| libcss.a<br>libcms.a<br>libcls.a | Single precision C library for small, medium or large memory model (compiler option **–F**)<br>(Some functions require the floating–point library. Also includes the startup code.) |
| libfps.a<br>libfpm.a<br>libfpl.a | Floating–point library (non–trapping) for each model |
| libfpst.a<br>libfpmt.a<br>libfplt.a | Floating–point library (trapping) for each model<br>(Control program option **––fp–trap**) |
| librts.a<br>librtm.a<br>librtl.a | Run–time library for each model |

*Table 2–1: Overview of M16C libraries*

## 2.2 LIBRARY FUNCTIONS

The tables in the sections below list all library functions, grouped per header file in which they are declared. Some functions are not completely implemented because their implementaion 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 CrossView Pro to simulate an I/O environment which enables you to debug your application.

### 2.2.1 ASSERT.H

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

### 2.2.2 COMPLEX.H

The complex number $z$ is also written as $x+yi$ where $x$ (the real part) and $y$ (the imaginary part) are real numbers of types `float`, `double` or `long double`. The real and imaginary part can be stored in structs or in arrays. This implementation uses arrays because structs may have different alignments.

The header file `complex.h` also defines the following macros for backward compatibility:

```
complex    _Complex    /* C99 keyword */
imaginary  _Imaginary  /* C99 keyword */
```

Parallel sets of functions are defined for double, float and long double. They are respectively named *function*, *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 divison and absolute value calculations which handles underflow, overflow and infinities with more care. The ISO/IEC 9899 `#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` | `casinh` | `cfasinhl` | Returns the complex arc hyperbolic sinus of `z`. |
| `cacosh` | `cacosh` | `cfacoshl` | Returns the complex arc hyperbolic cosinus 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 *z* raised to the power *w* ($z^w$) where both *z* and *w* 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* (respectively as a `double`, `float`, `long double`) |

### 2.2.3 CTYPE.H AND WCTYPE.H

The header file `ctype.h` declares the following functions which take a character *c* as an integer type argument. The header file `wctype.h` declares parallel wide–character functions which take a character *c* of the `wchar_t` type as argument.

| Ctype.h | Wctype.h | Description |
|---|---|---|
| `isalnum` | `iswalnum` | Returns a non–zero value when c is an alpha–betic character or a number ([A–Z][a–z][0–9]). |
| `isalpha` | `iswalpha` | Returns a non–zero value when c is an alphabetic character ([A–Z][a–z]). |
| `isblank` | `iswblank` | Returns a non–zero value when c is a blank character (tab, space...) |
| `iscntrl` | `iswcntrl` | Returns a non–zero value when c is a control character. |
| `isdigit` | `iswditit` | Returns a non–zero value when c is a numeric character ([0–9]). |
| `isgraph` | `iswgraph` | Returns a non–zero value when c is printable, but not a space. |
| `islower` | `iswlower` | Returns a non–zero value when c is a lowercase character ([a–z]). |
| `isprint` | `iswprint` | Returns a non–zero value when c is printable, including spaces. |
| `ispunct` | `iswpunct` | Returns a non–zero value when c is a punctuation character (such as '.', ',', '!'). |
| `isspace` | `iswspace` | Returns a non–zero value when c is a space type character (space, tab, vertical tab, formfeed, linefeed, carriage return). |
| `isupper` | `iswupper` | Returns a non–zero value when c is an uppercase character ([A–Z]). |
| `isxdigit` | `iswxdigit` | Returns a non–zero value when c is a hexadecimal digit ([0–9][A–F][a–f]). |

| Ctype.h | Wctype.h | Description |
|---|---|---|
| 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/IEC 9899. |
| _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/IEC 9899. |
| isascii | | Returns a non–zero value when c is in the range of 0 and 127. This function is not defined in ISO/IEC 9899. |
| toascii | | Converts c to an ASCII value (strip highest bit). This function is not defined in ISO/IEC 9899. |

### 2.2.4 ERRNO.H

`int errno` External variable that holds implementation defined error codes.

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

| | | |
|---|---|---|
| EZERO | 0 | No error |
| EPERM | 1 | Not owner |
| 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 prinff/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 |

***Error returned by file positioning routines***

| | | |
|---|---|---|
| ERR_POS | 29 | Positioning failure |

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

| | | |
|---|---|---|
| EILSEQ | 30 | Illegal byte sequence (including too few bytes) |

### 2.2.5 FCNTL.H

The file `fcntl.h` contains definitions of flags used by the low level function `_open()`. This header file is not defined in ISO/IEC9899.

### 2.2.6 FENV.H

Contains mechanisms to control the floating–point environment.

| | |
|---|---|
| `fegetenv` | Stores the current floating–point environment. |
| `feholdexept` | Saves the current floating–point environment and installs an environment that ignores all floating–point exceptions. |
| `fesetenv` | Restores a previously saved (fegetenv or feholdexcept) floating–point environment. |
| `feupdateenv` | Saves the currently raised floating–point exceptions, restores a previousely saved floating–point environment and finally raises the saved exceptions. |
| `feclearexcept` | Clears the current exception status flags corresponding to the flags specified in the argument. |
| `fegetexceptflag` | Stores the current setting of the floating–point status flags. |
| `feraiseexcept` | Raises the exceptions represented in the argument. As a result, other exceptions may be raised as well. |
| `fesetexceptflag` | Sets the current floating–point status flags. |
| `fetestexcept` | Returns the bitwise–OR of the exception macros corresponding to the exception flags which are currently set *and* are specified in the argument. |

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

```
FE_DIVBYZERO        FE_INEXACT          FE_INVALID
FE_OVERFLOW         FE_UNDERFLOW        FE_ALL_EXCEPT
```

| | |
|---|---|
| `fegetround` | Returns the current rounding direction, represented as one of the values of the rounding direction macros. |
| `fesetround` | Sets the current rounding directions. |

Currently no rounding mode macros are implemented.

### 2.2.7 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/IEC9899 standard been moved to the header file `math.h`. See also section 2.2.13, *Math.h and Tgmath.h*.

### 2.2.8 FSS.H

The header file `fss.h` contains definitions and prototypes for low level I/O functions used for CrossView Pro's file system simulation (fss). The low level functions are also declared in `stdio.h`; they are all implemented as fss functions. This header file is not defined in ISO/IEC9899.

| Stdio.h | Description |
|---|---|
| `_fss_break(void)` | Buffer and breakpoint functions for CrossView Pro. |
| `_fss_init(fd,is_close)` | Opens file descriptors 0, 1 and 2 and associates them with VIO stream 0 of CrossView Pro. |
| `_close(fd)`<br>`_lseek(fd,offset,whence)`<br>`_open(fd,flags)`<br>`_read(fd,*buff,cnt)`<br>`_unlink(*name)`<br>`_write(fd,*buffer,cnt)` | See *Low Level File Access Functions* in section 2.2.20, *Stdio.h*. |

### 2.2.9 INTTYPES.H AND STDINT.H

The header files **stdint.h** and **inttypes.h** provide additional declarations for integer types and have various characteristics. The **stdint.h** header file contains basic definitions of integer types of certain sizes, and corresponding sets of macros. This header file clearly refers to the corresponding sections in the ISO/IEC 9899 standard.
The **inttypes.h** header file incldues **stdint.h** and adds portable formatting and conversion functions. Below the conversion functions from **inttypes.h** are listed.

| | |
|---|---|
| `intmax_t imaxabs(intmax_t j);` | Returns the absolute value of j |
| `imaxdiv_t 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. |
| `intmax_t strtoimax(const char * restrict nptr, char ** restrict endptr, int base);` | Convert string to maximum sized integer. (Compare `strtol`) |
| `uintmax_t strtoumax(const char * restrict nptr, char ** restrict endptr, int base);` | Convert string to maximum sized unsigned integer. (Compare `strtoul`) |
| `intmax_t wcstoimax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);` | Convert wide string to maximum sized integer. (Compare `wctol`) |
| `uintmax_t wcstoumax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);` | Convert wide string to maximem sized unsigned integer. (Compare `wctoul`) |

### 2.2.10 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  ^=
```

### 2.2.11 LIMITS.H

Contains the sizes of integral types, defined as macros.

### 2.2.12 LOCALE.H

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

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

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

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

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

`struct lconv *localeconv( void )`

Returns a pointer to type `stuct 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.

## 2.2.13 MATH.H AND TGMATH.H

The header file `math.h` contains the prototypes for many mathematical functions. Before C99, all functions were computed using the double type (the float was automatically converted to double, prior to calculation). In this 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 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 sinus of `x`. |

| Math.h | | | Tgmath.h | Description |
|---|---|---|---|---|
| acosh | acoshf | acoshl | acosh | Returns the non-negative arc hyperbolic cosinus 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 $e^x$. |
| 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*) |

### *Rounding functions*

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

### *Remainder after devision*

| Math.h | | | Tgmath.h | Description |
|---|---|---|---|---|
| `fmod` | `fmodl` | `fmodf` | `fmod` | Returns the remainder `r` of `x-ny`. `n` is chosen as `trunc(x/y)`. `r` has the same sign as `x`. |
| `remainder` | `remainderl` `remainderf` | | `remainder` | Returns the remainder `r` of `x-ny`. `n` is chosen as `trunc(x/y)`. `r` may not have the same sign as `x`. (*Not implemented*) |
| `remquo` | `remquol` | `remquof` | `remquo` | Same as remainder. In addition, the argument `*quo` is given a specific value (see ISO). (*Not implemented*) |

### *frexp, ldexp, modf, scalbn, scalbln*

| Math.h | | | Tgmath.h | Description |
|---|---|---|---|---|
| `frexp` | `frexpl` | `frexpf` | `frexp` | Splits a float `x` into fraction *f* and exponent *n*, so that: $f = 0.0$ or $0.5 \leq \lvert f \rvert \leq 1.0$ and $f*2^n = x$. Returns *f*, stores n. |
| `ldexp` | `ldexpl` | `ldexpf` | `ldexp` | Inverse of `frexp`. Returns the result of $x*2^n$. (`x` and `n` are both arguments). |
| `modf` | `modfl` | `modff` | – | Splits a float `x` into fraction *f* and integer *n*, so that: $\lvert f \rvert < 1.0$ and $f+n=x$. Returns *f*, stores *n*. |
| `scalbn` | `scalbnl` | `scalbnf` | `scalbn` | Computes the result of $x*FLT\_RADIX^n$. efficiently, not normally by computing $FLT\_RADIX^n$ explicitly. |
| `scalbln` | `scalblnl` | `scalblnf` | `scalbln` | Same as `scalbn` but with argument `n` as `long int`. |

### *Power and absolute-value functions*

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

### *Manipulation functions: copysign, nan, nextafter, nexttoward*

| Math.h | | | Tgmath.h | Description |
|---|---|---|---|---|
| copysign | | copysignl<br>copysignf | copysign | Returns the value of x with the sign of y. |
| nan | nanl | nanf | – | Returns a quiet NaN, if available, with content indcated through *tagp*. (*Not implemented*) |
| nextafter | | nextafterl<br>nextafterf | 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 | | nexttowardl<br>nexttowardf | 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` | `fdiml` | `fdimf` | `fdim` | Returns the positive difference between: `\|x-y\|`. (*Not implemented*) |
| `fmax` | `fmaxl` | `fmaxf` | `fmax` | Returns the maximum value of their arguments. (*Not implemented*) |
| `fmin` | `fminl` | `fminf` | `fmin` | Returns the minimum value of their arguments. (*Not implemented*) |

### *Error and gamma (Not implemented)*

| Math.h | | | Tgmath.h | Description |
|---|---|---|---|---|
| `erf` | `erfl` | `erff` | `erf` | Computes the error function of x. (*Not implemented*) |
| `erfc` | `erfcl` | `erfcf` | `erc` | Computes the complementary error function of x. (*Not implemented*) |
| `lgamma` | `lgammal` | `lgammaf` | `lgamma` | Computes the $*\log_e\|\Gamma(x)\|$ (*Not implemented*) |
| `tgamma` | `tgammal` | `tgammaf` | `tgamma` | Computes $\Gamma(x)$ (*Not implemented*) |

### *Comparison macros*

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

| Math.h | Tgmath.h | Description |
|---|---|---|
| `isgreater` | – | Returns the value of `(x) > (y)` |
| `isgreaterequal` | – | Returns the value of `(x) >= (y)` |
| `isless` | – | Returns the value of `(x) < (y)` |
| `islessequal` | – | Returns the value of `(x) <= (y)` |
| `islessgreater` | – | Returns the value of `(x) < (y) \|\| (x) > (y)` |
| `isunordered` | – | Returns 1 if its arguments are unordered, 0 otherwise. |

### *Classification macros*

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

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

## 2.2.14 SETJMP.H

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

| | |
|---|---|
| `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()`. |

## 2.2.15 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 errouneous arithmetic operation (for example, zero devide, `overflow`) |
| `SIGSEGV` | 4 | An invalid access to storage |
| `SIGTERM` | 5 | A termination request sent to the program |
| `SIGABRT` | 6 | Abnormal terminiation, such as is initiated by the `abort` function. |

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

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

The next function determines how subsequent signals will be handled:

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

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

| | |
|---|---|
| `SIG_DFL` | Default behaviour is used |
| `SIG_IGN` | The signal is ignored |

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

## 2.2.16 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`. This header file contains the following macros:

| | |
|---|---|
| `va_arg(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_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 (ANSI specification). |
| `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. |

## 2.2.17 STDBOOL.H

This header file contains the following macro definitions. These names for boolean type and values are consisten 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
```

## 2.2.18 STDDEF.H

This header file defines the types for common use:

| | |
|---|---|
| `ptrdiff_t` | signed integer type of the result of subtracting two pointers. |
| `size_t` | unsigned integral type of the result of the `sizeof` operator. |
| `wchar_t` | integer type to represent character codes in large character sets. |

Besides these types, the following macros are defined:

| | |
|---|---|
| `NULL` | expands to the null pointer constant |
| `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`. |

## 2.2.19 STDINT.H

Section 2.2.9, *inttypes.h and stdint.h*

## 2.2.20 STDIO.H AND WCHAR.H

### *Types*

The header file **`stdio.h`** contains for performing input and output. A number of also have a parallel wide character function or macro, defined in **`wchar.h`**. The header file **`wchar.h`** also **`stdio.h`**.

In the C language, many I/O facilities are based on the concept of streams. The **`stdio.h`** header file defines the data type **`FILE`** which holds the information about a stream. An **`FILE`** object is created with the function fopen. The pointer to this object is used as an argument in many of the in this header file. The **`FILE`** object can contain the following information:

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

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

### *Macros*

| Stdio.h | Description |
|---|---|
| `BUFSIZ 512` | Size of the buffer used by the `setbuf`/`setvbuf` function: 512 |
| `EOF -1` | End of file indicator. |
| `WEOF UINTMAX` | End of file indicator.<br>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: 4<br>NOTE: According to ISO/IEC 9899 this value must be at least 8. |
| `FILENAME_MAX 100` | Maximum length of a filename: 100 |
| `_IOFBF`<br>`_IOLBF`<br>`_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 0x8000` | Maximum number of unique temporary filenames that can be generated: 0x8000 |
| `stderr`<br>`stdin`<br>`stdout` | Expressions of type ”pointer to FILE” that point to the FILE objects associated with standard error, input and output streams. |

### *Low level file access functions*

| Stdio.h | Description |
|---|---|
| `_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)* |

### *File access*

| Stdio.h | Description |
|---|---|
| `fopen(name,mode)` | Opens a file for a given mode. Available modes are:<br>"r" read; open text file for reading<br>"w" write; create text file for writing; if the file already exists its contents is discarded<br>"a" append; open existing text file or create new text file for writing at end of file<br>"r+" open text file for update; reading and writing<br>"w+" create text file for update; previous contents if any is discarded<br>"a+" append; open or create text file for update, writes at end of file<br>(*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)* |

| Stdio.h | Description |
|---|---|
| `freopen(`*`name`*`,`*`mode`*`,`*`stream`*`)` | Similar to fopen, but rather then 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:<br>`(void)setvbuf(`*`stream`*`,`*`buf`*`,_IOFBF, BUFSIZ).` |
| `setvbuf(`*`stream`*`,`*`buffer`*`, `*`mode`*`,`*`size`*`)` | Controls buffering for the *stream*; this function must be called before reading or writing. *Mode* can have the following values:<br>`_IOFBF` causes full buffering<br>`_IOLBF` causes line buffering of text files<br>`_IONBF` causes no buffering<br>If buffer is not NULL, it will be used as a buffer; otherwise a buffer will be allocated. *size* determines the buffer size. |

### *Character input/output*

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

- Flags (in any order):
  - `-` specifies left adjustment of the converted argument.
  - `+` a number is always preceded with a sign character. `+` has higher precedence than `space`.
  - `space` a negative number is preceded with a sign, positive numbers with a space.
  - `0` specifies padding to the field width with zeros (only for numbers).
  - `#` specifies an alternate output form. For o, the first digit will be zero. For x or X, "0x" and "0X" will be prefixed to the number. For e, E, f, g, G, the output always contains a decimal point, trailing zeros are not removed.

- A number specifying a minimum field width. The converted argument is printed in a field with at least the length specified here. If the converted argument has fewer characters than specified, it will be padded at the left side (or at the right when the flag '−' was specified) with spaces. Padding to numeric fields will be done with zeros when the flag '0' is also specified (only when padding left). Instead of a numeric value, also '*' may be specified, the value is then taken from the next argument, which is assumed to be of type int.
- A period. This separates the minimum field width from the precision.
- A number specifying the maximum length of a string to be printed. Or the number of digits printed after the decimal point (only for floating–point conversions). Or the minimum number of digits to be printed for an integer conversion. Instead of a numeric value, also '*' may be specified, the value is then taken from the next argument, which is assumed to be of type int.
- A length modifier 'h', 'l' or 'L'. 'h' indicates that the argument is to be treated as a short or unsigned short number. 'l' should be used if the argument is a long integer. 'L' indicates that the argument is a long double.

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 |

| Character | Printed as |
|---|---|
| n | int *, the number of characters written so far is written into the argument. This should be a pointer to an integer in default memory. No value is printed. |
| p | pointer (hexadecimal 24–bit value) |
| % | No argument is converted, a '%' is printed. |

*Table 2–2: Printf conversion characters*

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

The format string can contain :

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

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

- A '*', meaning that no assignment is done for this field.
- A number specifying the maximum field width.
- The conversion characters `d`, `i`, `n`, `o`, `u` and `x` may be preceede by 'h' if the argument is a pointer to `short` rather than `int`, or by 'l' (letter ell) if the argument is a pointer to `long`. The conversion characters `e`, `f`, and `g` may be preceede by 'l' if a pointer `double` rather than `float` is in the argument list, and by 'L' if 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 | float |
| e, E | float |
| g, G | float |
| n | int *, the number of characters written so far is written into the argument. No scanning is done. |
| p | pointer; hexadecimal 24–bit value which must be entered without 0x– prefix. |
| [...] | Matches a string of input characters from the set between the brackets. A NULL character is added to terminate the string. Specifying []...] includes the ’]’ character in the set of scanning characters. |
| [^...] | Matches a string of input characters not in the set between the brackets. A NULL character is added to terminate the string. Specifying [^]...] includes the ’]’ character in the set. |
| % | Literal ’%’, no assignment is done. |

*Table 2–3: Scanf conversion characters*

| Stdio.h | Wchar.h | Description |
|---|---|---|
| `fgetc(`*`stream`*`)` | `fgetwc(`*`stream`*`)` | Reads one character from *stream*. Returns the read character, or EOF/WEOF on error. (*FSS implementation)* |
| `getc(`*`stream`*`)` | `getwc(`*`stream`*`)` | Same as `fgetc`/`fgetwc` except that is implemented as a macro. (*FSS implementation)* NOTE: Currently #defined as `getchar()`/`getwchar()` because FILE I/O is not supported. Returns the read character, or EOF/WEOF on error. |
| `getchar(stdin)` | `getwchar(stdin)` | Reads one character from the `stdin` stream. Returns the character read or EOF/WEOF on error. Implemented as macro. (*FSS implementation)* |
| `fgets(*`*`s`*`,`*`n`*`, `*`stream`*`)` | `fgetws(*`*`s`*`,`*`n`*`, `*`stream`*`)` | Reads at most the next *n*–1 characters from the *stream* into array *s* until a newline is found. Returns s or NULL or EOF/WEOF on error. (*FSS implementation)* |
| `gets(*`*`s`*`,`*`n`*`,stdin)` | – | Reads at most the next *n*–1 characters from the `stdin` stream into array *s*. A newline is ignored. Returns *s* or NULL or EOF/WEOF on error. (*FSS implementation)* |
| `ungetc(`*`c`*`,`*`stream`*`)` | `ungetwc(`*`c`*`,`*`stream`*`)` | Pushes character *c* back onto the input *stream*. Returns EOF/WEOF on error. |
| `fscanf(`*`stream`*`, `*`format`*`,...)` | `fwscanf(`*`stream`*`, `*`format`*`,...)` | Performs a formatted read from the given *stream*. Returns the number of items converted succesfully. (*FSS implementation)* |

| Stdio.h | Wchar.h | Description |
|---|---|---|
| `scanf(format,...)` | `wscanf(format,...)` | Performs a formatted read from `stdin`. Returns the number of items converted succesfully. (*FSS implementation)* |
| `sscanf(*s, format,...)` | `swscanf(*s, format,...)` | Performs a formatted read from the string *s*. Returns the number of items converted succesfully. |
| `vfscanf(stream, format,arg)` | `vfwscanf(stream, format,arg)` | Same as `fscanf`/`fwscanf`, but extra arguments are given as variable argument list *arg*. (See section 2.2.16, *stdarg.h*) |
| `vscanf(format,arg)` | `vwscanf(format, arg)` | Same as `sscanf`/`swscanf`, but extra arguments are given as variable argument list *arg*. (See section 2.2.16, *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 2.2.16, *stdarg.h*) |
| `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)* |

| Stdio.h | Wchar.h | Description |
|---|---|---|
| puts(*s) | – | Writes string *s* to the stdout stream. Returns EOF/WEOF on error. (*FSS implementation)* |
| 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 2.2.16, *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 2.2.16, *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 2.2.16, *stdarg.h*) |

### *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 succesfully 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 succesfully 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:<br>`(void) fseek( stream, 0L, SEEK_SET );`<br>`clearerr( stream );`<br>(*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 succesful. |
| `rename(`*`old`*`,`*`new`*`)` | Changes the name of the file from old name to new name. Returns a non–zero value if not succesful. |
| `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 2.2.4, *errno.h*) |

## 2.2.21 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
- Envirnoment communication
- Searching and sorting

- Integer arithmetic
- Multibyte/wide character and string conversions.

### *Macros*

| | |
|---|---|
| `RAND_MAX 32767` | Highest number that can be returned by the `rand`/`srand` function. |
| `EXIT_SUCCES 0`<br>`EXIT_FAILURE 1` | Predefined exit codes that can be used in the `exit` 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 2.2.12, *locale.h*). |

### *Numeric conversions*

Next convert the intial 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)
```

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

**Stdlib.h**

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

**Wchar.h**

```
float       wcstof(*s,**endp)
double      wcstod(*s,**endp)
long double wcstold(*s,**endp)
```

Next convert the initial portion of the string *s to a `long`, `long long`, `unsigned long` and `unsigned long long` respectively. Base specifies the radix. `*endp` will point to the first character not used by the conversion.

**Stdlib.h**

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

**Wchar.h**

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

### *Random number generation*

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

### *Memory management*

| | |
|---|---|
| `malloc(size)` | Allocates space for an object with size *size*. The allocated space is not initialized. Returns a pointer to the allocated space. |
| `calloc(`*nobj*`,`*size*`)` | Allocates space for n objects with size *size*. The allocated space is initialized with zeros. Returns a pointer to the allocated space. |
| `free(*ptr)` | Deallocates the memory space pointed to by *ptr* which should be a pointer earlier returned by the `malloc` or `calloc` function. |
| `realloc(*ptr,`*size*`)` | Deallocates the old object pointed to by ptr and returns a pointer to a niew object with size *size*. The new object cannot have a size larger than the previous object. |

### *Environment communication*

| | |
|---|---|
| `abort()` | Causes abnormal program termination. If the signal SIGABRTis caught, the signal handler may take over control. (See section 2.2.15, *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 no registered by the `atexit` function or signal handlers registerd by the `signal` function are called. |
| `getenv(*s)` | Searches an environment list for a string *s*. Returns a pointer to the contents of *s*.<br>NOTE: this function is not implemented because there is no OS. |
| `system(*s)` | Passes the string *s* to the environment for execution.<br>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` `long` `long long` | `abs(`*j*`)` `labs(`*j*`)` `llabs(`*j*`)` | Compute the absolute value of an `int`, `long int`, and `long long int` *j* resepectively. |
| `div_t` `ldiv_t` `lldiv_t` | `div(`*x*`,`*y*`)` `ldiv(`*x*`,`*y*`)` `lldiv(`*x*`,`*y*`)` | Compute *x*/*y* and *x*%*y* in a single operation. *X* and *y* have respectively type `int`, `long int` and `long long int`. The result is stored in the members `quot` and `rem` of `struct div_t`, `ldiv_t` and `lldiv_t` which have the same types. |

### *Multibyte/wide character and string conversions*

| | |
|---|---|
| `mblen(*`*s*`,`*n*`)` | Determines the number of bytes in the multi–byte character pointed to by *s*. At most *n* characters will be examined. (See also `mbrlen` in section 2.2.25, *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 2.2.25, *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 2.2.25, *wchar.h*) |

## 2.2.22 STRING.H AND WCHAR.H

This header file provides numerous functions for manipulating strings. By convention, strings in C are arrays of characters with a terminating null character. Most functions therefore take arguments of type `*char`. However, many functions have also parallel wide–character functions which take arguments of type `*wchar_t`. These functions are declared in `wchar.h`.

### *Copying and concatenation functions*

| Stdio.h | Wchar.h | Description |
|---|---|---|
| `memcpy(*s1,*s2,n)` | `wmemcpy(*s1,*s2,n)` | Copies *n* characters from **s2* into **s1* and returns *s1. If *s1 and *s2 overlap the result is undefined. |
| `memmove(*s1,*s2,n)` | `wmemmove(*s1,*s2,n)` | Same as `memcpy`, but overlapping strings are handled correctly. Returns **s1*. |
| `strcpy(*s1,*s2)` | `wcscpy(*s1,*s2)` | Copies **s2* into **s1* and returns **s1*. If **s1* and **s2* overlap the result is undefined. |
| `strncpy(*s1,*s2,n)` | `wcsncpy(*s1,*s2,n)` | Copies not more than *n* characters from **s2* into **s1* and returns **s1*. If **s1* and **s2* overlap the result is undefined. |
| `strcat(*s1,*s2)` | `wcscat(*s1,*s2)` | Appends a copy of **s2* to **s1* and returns **s1*. If **s1* and **s2* overlap the result is undefined. |
| `strncat(*s1,*s2,n)` | `wcsncat(*s1,*s2,n)` | Appends not more than *n* characters from **s2* to **s1* and returns **s1*. If **s1* and **s2* overlap the result is undefined. |

### *Comparison functions*

| Stdio.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 2.2.12, *locale.h*) |
| `strxfrm(*s1,*s2,n)` | `wcsxfrm(*s1,*s2,n)` | Transforms (a local) string **s2* so that a comparison between transformed strings with `strcmp` gives the same result as a comparison between non–transformed strings with `strcoll`. Returns the transformed string **s1*. |

### *Search functions*

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

***Miscellaneous functions***

| Stdio.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 2.2.4, *errno.h*) |
| `strlen(*s)` | `wcslen(*s)` | Returns the length of string **s*. |

## 2.2.23 TIME.H AND WCHAR.H

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

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

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

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

### *Time manipulation*

| | |
|---|---|
| `clock` | Returns the application's best approximation to the processor time used by the program since it was started. This low–level routine is not implemented because it strongly depends on the hardware. To determine the time in seconds, the result of clock should be divided by the value defined as<br>`CLOCKS_PER_SEC 12000000` |
| `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`:

| Stdio.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 2.2.12, *locale.h*). You can use the next conversion specifiers:

%a abbreviated weekday name
%A full weekday name
%b abbreviated month name
%B full month name
%c local date and time representation
%d day of the month (01–31)
%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)
%p local equivalent of AM or PM
%S second (00–59)
%U week number of the year, Sunday as first day of the week (00–53)
%w weekday (0–6, Sunday is 0)
%W week number of the year, Monday as first day of the week (00–53)
%x local date representation
%X local time representation
%y year without century (00–99)
%Y year with century
%Z time zone name, if any
%% %

### 2.2.24 UNISTD.H

The file **unistd.h** contains standard UNIX I/O functions. These functions are all implemented using CrossView Pro's file system simulation. This header file is not defined in ISO/IEC9899.

| | |
|---|---|
| `access(*name,mode)` | Use the file system simulation of CrossView Pro 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:<br>`R_OK` Checks read permission.<br>`W_OK` Checks write permission.<br>`X_OK` Checks execute (search) permission.<br>`F_OK` Checks to see if the file exists.<br>(*FSS implementation)* |
| `chdir(*path)` | Use the file system simulation feature of CrossView Pro to change the current directory on the host to the directory indicated by *path*.<br>(*FSS implementation)* |
| `close(fd)` | File close function. The given file descriptor should be properly closed. This function calls `_close()`.<br>(*FSS implementation)* |
| `getcwd(*buf,size)` | Use the file system simulation feature of CrossView Pro 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()`.<br>(*FSS implementation)* |
| `read(fd,*buff,cnt)` | Reads a sequence of characters from a file. This function calls `_read()`. (*FSS implementation)* |
| `stat(*name,*buff)` | Use the file system simulation feature of CrossView Pro to stat() a file on the host platform.<br>(*FSS implementation)* |
| `unlink(*name)` | Removes the named file, so that a subsequent attempt to open it fails. Calls `_unlink()`.<br>(*FSS implementation)* |
| `write(fd,*buff,cnt)` | Write a sequence of characters to a file. Calls `_write()`. (*FSS implementation)* |

## 2.2.25 WCHAR.H

Many in `wchar.h` represent the wide–character variant of other so these are discussed together. (See sections 2.2.20, *stdio.h*, 2.2.21, *stdlib.h*, 2.2.22, *strings.h* and 2.2.23, *time.h*).

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

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

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

| | |
|---|---|
| `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 2.2.21, *stdlib.h*. The initial conversion state is specified by *ps*. The input sequence of multibyte charactersis specified indirectly by *src*. |
| `wcsrtombs(*s,**src,n,*ps)` | Restartable version of `wcstombs`. See section 2.2.21, *stdlib.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 2.2.21, *stdlib*. |

| | |
|---|---|
| `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*. |

## 2.2.26 WCTYPE.H

Most in **`wctype.h`** represent the wide-character variant of declared in **`ctype.h`** and are discussed in section 2.2.3, *ctype.h*. In addition, this header file provides extensible, locale specific, wide character classification.

| | |
|---|---|
| `wctype(*property)` | Constructs a value of type `wctype_t` that describes a class of wide characters identified by the string ***property*. If property identifies a valid class of wide characters according to the LC_TYPE category (see 2.2.12, *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"))` |

| Function | Equivalent to locale specific test |
|---|---|
| `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 2.2.12, *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")` |

CHAPTER 3

# ASSEMBLY LANGUAGE

TASKING

# CHAPTER 3

## 3.1 INTRODUCTION

This chapter contains a detailed description of all built–in assembly functions directives and controls. For a description of the M16C instruction set, refer to the *M16C Series Software Manual* [Renesas].

## 3.2 BUILT-IN ASSEMBLY FUNCTIONS

### 3.2.1 OVERVIEW OF BUILT-IN ASSEMBLY FUNCTIONS

The built–in assembler functions are grouped into the following types:

- **Mathematical functions** comprise, among others, transcendental, random value, and min/max functions.
- **String functions** compare strings, return the length of a string, and return the position of a substring within a string.
- **Macro functions** return information about macros.
- **Address calculation functions** return the high or low part of an address.
- **Assembler mode functions** relating assembler operation.

The following tables provide an overview of all built–in assembler functions. *expr* can be any assembly expression resulting in an integer value. Expressions are explained in section 4.6, *Assembly Expressions*, in chapter *Assembly Language* of the *User's Guide*.

***Overview of mathematical functions***

| Function | Description |
|---|---|
| @ABS(*expr*) | Absolute value |
| @MAX(*expr*,[,...,*exprN*]) | Maximum value |
| @MIN(*expr*,[,...,*exprN*]) | Minimum value |
| @SGN(*expr*) | Returns the sign of an expression as –1, 0 or 1 |

### *Overview of string functions*

| Function | Description |
|---|---|
| @CAT(*str1*,*str2*) | Concatenate strings |
| @LEN(*string*) | Length of string |
| @POS(*str1*,*str2*[,*start*]) | Position of substring in string |
| @SCP(*str1*,*str2*) | Returns 1 if two strings are equal |
| @SUB(*str1*,*expr*,*expr*) | Returns substring in string |

### *Overview of macro functions*

| Function | Description |
|---|---|
| @ARG('*symbol*'\|*expr*) | Test if macro argument is present |
| @CNT() | Return number of macro arguments |
| @MAC(*symbol*) | Test if symbol is defined as a macro |
| @MXP() | Test if macro expansion is active |

### *Overview of address calculation functions*

| Function | Description |
|---|---|
| @LSW(*expr*) | Returns lower 16 bits of expression value |
| @MSW(*expr*) | Returns bits 16..31 of expression value |

### *Overview of assembler mode functions*

| Function | Description |
|---|---|
| @DEF('*symbol*'\|*symbol*) | Returns 1 if symbol has been defined |
| @LST() | LIST control flag value |

### 3.2.2 DETAILED DESCRIPTION OF BUILT-IN ASSEMBLY FUNCTIONS

***@ABS(****expression****)***

Returns the absolute value of *expression* as an integer value.

Example:

```
MOV.W #@ABS(VAL), R0 ;load absolute value into R0
```

***@ARG('****symbol****'*** | *expression****)***

Returns an integer 1 if the macro argument represented by *symbol* or *expression* is present, 0 otherwise. If the argument is a symbol it must be single–quoted and refer to a formal argument name. If the argument is an *expression* it refers to the ordinal position of the argument in the macro formal argument list. The assembler issues a warning if this function is used when no macro expansion is active.

Example:

```
IF @ARG('TWIDDLE')   ;twiddle factor provided?
IF @ARG(1)           ;is first argument present
```

***@CAT(****string1****,****string2****)***

Concatenates the two strings into one string. The two strings must be enclosed in single or double quotes.

Example:

```
DEFINE ID "@CAT('M1','6C')"  ;ID = 'M16C'
```

***@CNT()***

Returns the number of arguments of the current macro expansion as an integer. The assembler issues a warning if this function is used when no macro expansion is active.

Example:

```
ARGCNT SET @CNT()    ;reserve argument count
```

***@DEF('**symbol**' | **symbol**)***

Returns an integer 1 if *symbol* has been defined, 0 otherwise. *symbol* can be any symbol or label not associated with a `MACRO` or `DEFSECT` directive. 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 or label.

Example:

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

***@LEN(**string**)***

Returns the length of *string* as an integer.

Example:

```
SLEN SET @LEN('Altium')     ;SLEN = 6
```

***@LST()***

Returns the value of the `$LIST ON/OFF` control flag as an integer. Each time a `$LIST ON` control is encountered in the assembly source, the flag is incremented. Each time a `$LIST OFF` control is encountered, the flag is decremented.

Example:

```
DUP  @ABS(@LST())           ;list unconditionally
```

***@LSW(**expression**)***

Returns the lower 16 bits of a value. `@LSW(`*`expression`*`)` is equivalent to `(`*`expression`*` & 0xffff)`.

Example:

```
mov.w  #@LSW(COUNT),a0      ;lower 16 bits of COUNT
```

***@MAC(**symbol**)***

Returns an integer 1 if *symbol* has been defined as a macro name, 0 otherwise.

Example:

```
IF  @MAC(DOMUL)             ;does macro DOMUL exist?
```

***@MAX(****expr1[,exprN]...****)***

Returns the largest of *expr1*,...,*exprN* as an integer.

Example:

```
MAX: DB  @MAX(1,5,-3)        ;MAX = 5
```

***@MIN(****expr1[,exprN]...****)***

Returns the smallest of *expr1*,...,*exprN* as an integer.

Example:

```
MIN: DB  @MIN(1,5,-3)        ;Min = -3
```

***@MSW(****expression****)***

Returns bits 16..31 of a value. `@MSW(`*`expression`*`)` is equivalent to `((`*`expression`*`>>16) & 0xffff)`.

Example:

```
movw.w  #@MSW(COUNT),a0   ;bits 16..31 of COUNT
```

***@MXP()***

Returns an integer 1 if the assembler is expanding a macro, 0 otherwise.

Example:

```
IF  @MXP()                    ;macro expansion active?
```

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

Returns the position of *string2* in *string1* as an integer, starting at position *start*. If *start* is not given the search begins at the beginning of *string1*. If the *start* argument is specified it must be a positive integer and cannot exceed the length of the source string. Note that the first position in a string is position 0.

Example:

```
ID   EQU  @POS('ASMFUNCTION','FUNC') ;ID  = 3
ID2  EQU  @POS('ABCDABCD','B',2)     ;ID2 = 5
```

***@SCP(****string1,string2****)***

Returns an integer 1 if the two strings are equal, 0 otherwise. The two strings must be separated by a comma.

Example:

```
STR SET 'MAIN'
IF  @SCP(STR,'MAIN')     ;does STR equal MAIN? yes
```

***@SGN(****expression****)***

Returns the sign of *expression* as an integer: –1 if the argument is negative, 0 if zero, 1 if positive. The *expression* can be relative or absolute.

Example:

```
VAR1  SET @SGN(-12)      ;VAR1 = -1
VAR2  SET @SGN(0)        ;VAR2 =  0
VAR3  SET @SGN(28)       ;VAR3 =  1
```

***@SUB(****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 "@SUB('ASMFUNCTION',3,4)"  ;ID = 'FUNC'
```

## 3.3 ASSEMBLER DIRECTIVES AND CONTROLS

### 3.3.1 OVERVIEW OF ASSEMBLER DIRECTIVES

Assembler directives are grouped in the following categories:

- Assembly control directives
- Symbol definition directives
- Data definition / Storage allocation directives
- Macro and conditional assembly directives
- Debug directives

The following tables provide an overview of all assembler directives.

***Overview of assembly control directives***

| Directive | Description |
|---|---|
| COMMENT | Start comment lines |
| DEFINE | Define substitution string |
| DEFSECT | Define section name and attributes |
| END | End of source program |
| FAIL | Programmer generated error message |
| INCLUDE | Include secondary file |
| MSG | Programmer generated message |
| RADIX | Change input radix for constants |
| SECT | Activate a declared section |
| UNDEF | Undefine DEFINE symbol |
| WARN | Programmer generated warning |

### *Overview of symbol definition directives*

| Directive | Description |
|---|---|
| BTEQU | Bit equate |
| EQU | Assign permanent value to a symbol |
| EXTERN | External symbol declaration |
| GLOBAL | Global section symbol declaration |
| LOCAL | Local symbol declaration |
| SET | Assign value to a symbol |
| SIZE | Set size of symbol in the ELF symbol table |
| TYPE | Set symbol type in the ELF symbol table |
| WEAK | Mark symbol as 'weak' |

### *Overview of data definition / storage allocation directives*

| Directive | Description |
|---|---|
| ALIGN | Define alignment |
| ASCII / ASCIZ | Define ASCII string without / with ending NULL byte |
| BS | Define block storage (initialized) |
| BSB | Define byte block storage (initialized) |
| BSBIT | Define bit block storage in bit addressable data |
| BSL | Define long block storage (initialized) |
| BSW | Define word block storage (initialized) |
| DB | Define constant byte |
| DBIT | Define constant bit |
| DL | Define a constant long (4 bytes) |
| DS | Define storage |
| DW | Define a constant word (2 bytes) |
| FLOAT / DOUBLE | Define a constant float or double |

***Overview of macro and conditional assembly directives***

| Directive | Description |
|---|---|
| DUP / ENDM | Duplicate sequence of source lines |
| DUPA / ENDM | Duplicate sequence with arguments |
| DUPC / ENDM | Duplicate sequence with characters |
| DUPF / ENDM | Duplicate sequence in loop |
| EXITM | Exit macro |
| IF / ELIF / ELSE / ENDIF | Conditional assembly |
| MACRO / ENDM | Define macro |
| PMACRO | Undefine (purge) macro |

***Overview of debug directives***

| Function | Description |
|---|---|
| CALLS | Passes call information to object file. Used by the linker to build a call graph and calculate stack size |

## 3.3.2 DETAILED DESCRIPTION OF ASSEMBLER DIRECTIVES

Some assembler directives can be preceeded with a label. If you do not preceede an assembler directive with a label, you must use white space instead (spaces or tabs). The assembler recognizes both upper and lower case for directives.

# ALIGN

## Syntax

**ALIGN** *expression*

## Description

With the `ALIGN` directive you instruct the assembler to align the location counter. By default the assembler aligns on one byte (or bit in bit–type sections).

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.

## Example

```
    DEFSECT "code", code
    SECT "code"
    ALIGN 4       ;the assembler aligns
    add.w a0,r0   ;this instruction on 4 bytes

    ALIGN 6       ;not a 2^k value.
lab1:             ;a warning is issued
                  ;lab1 is aligned on 8 bytes
```

## Related information

–

# ASCII/ASCIZ

## Syntax

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

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

## Description

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

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

## Example

```
STRING:   ASCII  "Hello world"

STRINGZ:  ASCIZ  "Hello world"
```

With the `DB` directive you can obain exactly the same effect:

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

## Related information

**DS** (Define storage)

**DB** (Define constant byte)

# BS

## Syntax

[*label*] **BS** *expression1*[,*expression2*]

## Description

With the `BS` directive (Block Storage) the assembler reserves a block of memory.

With *expression1* you specify the number of bits, or bytes (depending on the MAU size of a section) you want to reserve, and how much the location counter will advance. The expression must be an integer greater than zero and cannot contain any forward references to address labels (labels that have not yet been defined).

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

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

Initialization of a block of memory only happens in sections with section attribute `init` or `romdata`. In other sections, the assembler issues a warning and only reserves space, just as with DS.

## Example

The BS directive is for example useful to define an array that is only partially initialized:

```
DEFSECT "test_INI_DA", data, init
SECT "test_INI_DA"
DB 84,101,115,116  ; initalize 4 bytes
BS 96              ; reserve another 96 bytes (zeroed)
```

**BSB** (Define byte block storage (initialized))
**BSBIT** (Define bit block storage (initialized))
**BSL** (Define long block storage (initialized))
**BSW** (Define word block storage (initialized))

**DS** (Define storage)

# BSB

## Syntax

[*label*] **BSB** *expression1*[,*expression2*]

## Description

With the `BSB` directive (Byte Block Storage) the assembler reserves a block of bytes in memory.

With *expression1* you specify the number of bytes you want to reserve, and how much the location counter will advance. The expression must be an integer greater than zero and cannot contain any forward references to address labels (labels that have not yet been defined).

With *expression2* you can specify a value to initialize the block with. Only the least significant byte (or bit in bit sections) of *expression2* is used. If you omit *expression2*, the default is zero.

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

Initialization of a block of memory only happens in sections with attribute `init` or `romdata`. In other sections, the assembler issues a warning and only reserves space, just as with DS.

## Example

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

```
DEFSECT "test_INI_DA", data, init
SECT "test_INI_DA"
DB 84,101,115,116  ; initalize 4 bytes
BSB 96,0xFF        ; reserve another 96 bytes,
                     initialized with FF.
```

**BS** (Block storage)

**DS** (Define storage)

# BSBIT

## Syntax

[*label*] **BSBIT** *expression1*[,*expression2*]

## Description

With the `BSBIT` directive (Bit Block Storage) the assembler reserves a block of bits in memory.

With *expression1* you specify the number of bits you want to reserve, and how much the location counter will advance. The expression must be an integer greater than zero and cannot contain any forward references to address labels (labels that have not yet been defined).

With *expression2* you can specify a value (0 or 1) to initialize the bits with. Only the least significant bit of *expression2* is used. If you omit *expression2*, the default is zero.

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

You can use the `BSBIT` directive only within bit sections.

Initialization of a block of memory only happens in sections with attribute `init` or `romdata`. In other sections, the assembler issues a warning and only reserves space, just as with DS.

## Example

To initialize 16 bits with the value '1':

```
DEFSECT "test_INI_BI", bit, init
SECT "test_INI_BI"
BSBIT 16,1      ; reserve 16 bits, initialized with '1'
```

**BS** (Block storage)

**DS** (Define storage)

# BSL/BSW

## Syntax

[*label*] **BSL** *expression1*[,*expression2*]

[*label*] **BSW** *expression1*[,*expression2*]

## Description

With the `BSL` or `BSW` directive the assembler reserves a block of longs (32 bits) or words (16 bits) in memory.

With *expression1* you specify the number of longs or words you want to reserve, and how much the location counter will advance. The expression must be an integer greater than zero and cannot contain any forward references to address labels (labels that have not yet been defined).

With *expression2* you can specify a value to initialize the block with. Only the least significant long / word (or bit in bit sections) of *expression2* is used. If you omit *expression2*, the default is zero.

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

Initialization of a block of memory only happens in sections with attribute `init` or `romdata`. In other sections, the assembler issues a warning and only reserves space, just as with DS.

## Examples

```
LNG:   BSL 16,0x12345678  ; initalized with 0x12345678

WRD1:  BSW 16,0x1234      ; initalized with 0x1234
WRD2:  BSW 16,0x12345678  ; initalized with 0x5678
```

You can of course initialize a single long or word imitating the effect of the `DL/DW` directive:

```
LNG:  BSL 1,0x12345678
```

has the same effect as:

```
LNG:  DL  0x78563412
```

**BS** (Block storage)

**DS** (Define storage)

# BTEQU

## Syntax

*symbol* **BTEQU** *bit*,*base*

## Description

With the `BTEQU` directive (equate symbol to a bit value) you can assign a bit position to a *symbol*. The symbol name cannot be redefined anywhere else in the program.

*Base* is the base address in which you want to identify a *bit*. You then can use *symbol* to refer to that bit.

## Example

```
Flp_Bit BTEQU 5,19  ;bit 5 in byte 19
```

The symbol `Flp_Bit` is now associated with the forementioned bit and you can use the symbol for example to clear the bit:

```
bclr Flp_Bit
```

## Related information

–

# CALLS

## Syntax

**CALLS** '*caller*'**,** '*callee*' [**,***call_frequency* [**,***stack_usage*]...]

## Description

Create a flow graph reference between *caller* and *callee*. With this information the linker can build a call graph and calculate stack size. *Caller* and *Callee* are names of functions. The *call_frequency* shows how many times the *callee* is called. The *stack_usage* represents the stack usage in bytes at the location of the call or the maximum stack usage of function *caller*. A function can use multiple stacks.

The compiler inserts `CALLS` directives automatically to pass call tree information. The compiler emits the name `__INDIRECT__` to indicate an indirect call. Normally it is not necessary to use the `CALLS` directive in hand coded assembly.

A label is not allowed before this directive.

## Example

```
CALLS '_main', '_nfunc', 1, 5
```

Indicates that the function `_main` calls the function `_nfunc` 1 time and that the stack usage at the location of the call is 5 bytes.

```
CALLS '_main', '', 0, 5
```

Specifies the maximum stack usage of function `_main` (5 bytes).

```
CALLS '_f', '__INDIRECT__', 2, 3, 4
```

Indicates that the function `_f` executes an indirect call 2 times and that the stack usages at the location of the call is 3 and 4 bytes.

```
CALLS '__INDIRECT__', '_f'
```

Indicates that the address of function `_f` is taken.

## Related information

–

# COMMENT

## Syntax

**COMMENT** *delimiter*
.
.
*delimiter*

## Description

With the `COMMENT` directive (Start Comment Lines) you can define one or more lines as comments. The first non–blank character after the `.COMMENT` directive is the comment delimiter. The two delimiters are used to define the comment text. The line containing the second comment delimiter will be considered the last line of the comment. The comment text can include any printable characters and the comment text will be produced in the source listing as it appears in the source file.

A label is not allowed before this directive.

## Example

```
COMMENT  + This is a one line comment +
COMMENT  * This is a multiple line
           comment. Any number of lines
           can be placed between the two
           delimiters.
         *
```

## Related information


–

# DB

## Syntax

[*label*] **DB** *argument*[**,***argument*]...

## Description

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

An *argument* can be:

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

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

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

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

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

```
DB 'R'          ; = 0x52
DB 'AB',,'D'    ; = 0x41420043
```

## Example

```
TABLE:  DB  'two',0,'strings',0
CHARS:  DB  'A','B','C','D'
```

## Related information

**BS** (Block storage)

**DS** (Define storage)

# DBIT

## Syntax

[*label*] **DBIT** *expression*[,*expression*]...

## Description

With the `DBIT` directive (Define Bit) you allocate and initialize memory in bit units for each *expression*.

You can use the `DBIT` directive only within sections of the type `bit`.

An *expression* can be any expression resulting in 0 or 1.

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

## Example

```
NBITS:  DBIT 1,0,1,1     ; allocate and initialize
                         ; four bits.
```

## Related information

**BS** (Block storage)

**DS** (Define storage)

# DEFINE

## Syntax

**DEFINE** *symbol string*

## Description

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

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

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

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

A label is not allowed before this directive.

## Example

If the following `DEFINE` directive occurred in the first part of the source program:

```
DEFINE  LEN  '32'
```

then the source lines below:

```
DS   LEN
MSG  "The length is: LEN"
```

would be transformed by the assembler to the following:

```
DS   32
MSG  "The length is: 32"
```

## Related information

**UNDEF** (Undefine DEFINE symbol)
**SET** (Set temporary value to a symbol)

# DEFSECT

## Syntax

**DEFSECT "***name***",** *type* [**,** *attribute* ]... [**AT** *address*]

## Description

With the `DEFSECT` directive you can define a section with a *name*, *type* and optional *attributes*. Before any code or data can be placed in a section, you must use the `SECT` directive to activate the section.

You can specify the following section types:

| Section Type | Description | Space |
|---|---|---|
| code | Code section | far |
| data | __near data section | near |
| fdata | __far data section | far |
| bit | __bit type section | bit |
| bita | __bita type section (bitaddressable) | bita |

Sections of a specified type are located by the linker in a memory space as shown in the table above. The space names are defined in a so–called 'linker script file' (files with the extension `.lsl`) delivered with the product in the directory `include.lsl`.

You can specify the following section attributes:

| Attribute | Description | Allowed on type |
|---|---|---|
| at *address* | Locate the section at the specified address | all |
| clear | Clear the section at program startup | all except: code |
| noclear | Do not clear the section at program startup (default) | all except: code |
| init | Section is located in ROM. During program initialization the section is copied from ROM to RAM. | all |

| Attribute | Description | Allowed on type |
|---|---|---|
| max | When 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. | all except: code |
| romdata | Indicates that the section contains data to be placed in ROM | all |
| fit *expression* | Indicates that the section may not cross a given boundary. As a result, the specified size is also the maximum possible size for such a section. | all |

A label is not allowed before this directive.

### Examples

```
DEFSECT  "text_DA", DATA   ;declare section text_DA
SECT     "text_DA"         ;switch to section text_DA
```

**SECT** (Activate a declared section)

Section 4.9 *Working with Sections* in chapter *Assembly Language* of the *User's Guide*.

# DL/DW

## Syntax

[*label*] **DL** *argument*[**,***argument*]...

[*label*] **DW** *argument*[**,***argument*]...

## Description

With the `DL` or `DW` directive the assembler allocates and initializes a long (32 bits) or a word (16 bits) of memory for each *argument*.

An *argument* can be:

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

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

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

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

```
DL 'R'          ; = 0x52000000
DL 'ABCD'       ; = 0x44434241

DW 'R'          ; = 0x5200
DW 'AB'         ; = 0x4241
DW 'ABCD'       ; = 0x4241   ;value truncated
```

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

## Examples

```
LNG:  DL  14,1635,0x34266243,'ABCD'

WRD:  DW  14,1635,0x2662,'AB'
```

With the `DB` directive you can obain exactly the same effect:

```
LNG:   DB   14,0,0,0,1635%256,6,0,0,
                0x43,0x62,0x26,0x34,'D','C','B','A'

WRD:   DB   14,0,1635%256,6,0x62,0x26,'B','A'
```

### Related information

**BS** (Block storage)

**DS** (Define storage)

# DS

## Syntax

[*label*] **DS** *expression*

## Description

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

With *expression* you specify the number of bits, bytes (depending on the mau size of a the section) you want to reserve, and how much the location counter will advance.

The expression must be an integer greater than zero and cannot contain any forward references to address labels (labels that have not yet been defined).

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

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

## Example

To reserve 12 bytes (not initialized) of memory in a data section:

```
S_BUF  DS  12     ; Sample buffer
```

## Related information

**BS** (Block storage)

**DB** (Define constant byte)
**DBIT** (Define constant bit)
**DL** (Define constant long)
**DW** (Define constant word)

**ASCII** / **ASCIZ** (Define ASCII string)

# DUP / ENDM

## Syntax

[*label*] **DUP** *expression*
.
.
**ENDM**

## Description

The sequence of source lines between the `DUP` and `ENDM` directives will be duplicated by the number specified by the integer *expression*. 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 to address labels (labels that have not already been defined). You can nest the `DUP` directive to any level.

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

## Example

Consider the following source input statements,

```
COUNT   SET   3
        DUP   COUNT     ; duplicate NOP count times
        NOP
        ENDM
```

This is expanded as follows:

```
COUNT   SET   3
        NOP
        NOP
        NOP
```

## Related information

**DUPA** (Duplicate Sequence with Arguments),
**DUPC** (Duplicate Sequence with Characters),
**DUPF** (Duplicate Sequence in Loop),
**MACRO** (Define Macro)

# DUPA / ENDM

## Syntax

[*label*] **DUPA** *formal_arg*,*argument*[,*argument*]...
.
.
**ENDM**

## Description

With the `DUPA` and `ENDM` directives (Duplicate Sequence with Arguments) you can repeat a block of source statements for each *argument*. For each repetition, every occurrence of the *formal_arg* parameter within the block is replaced with each succeeding *argument* string. If an argument includes an embedded blank or other assembler–significant character, it must be enclosed with single quotes.

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

## Example

Consider the following source input statements,

```
DUPA  VALUE,12,,32,34
DB  VALUE
ENDM
```

This is expanded as follows:

```
DB  12
DB  VALUE  ; results in a warning
DB  32
DB  34
```

The second statement results in a warning of the assembler that the local symbol `VALUE` is not defined in this module and is made external.

## Related information

**DUP** (Duplicate Sequence of Source Lines),
**DUPC** (Duplicate Sequence with Characters),
**DUPF** (Duplicate Sequence in Loop),
**MACRO** (Define Macro)

# DUPC / ENDM

## Syntax

[*label*] **DUPC** *formal_arg*,*string*
.
.
**ENDM**

## Description

With the `DUPC` and `ENDM` directives (Duplicate Sequence with Characters) you can repeat a block of source statements for each character within *string*. For each character in the *string*, the *formal_arg* parameter within the block is replaced with that character If the *string* is empty, then the block is skipped.

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

## Example

Consider the following source input statements,

```
DUPC  VALUE,'123'
DB  VALUE
ENDM
```

This is expanded as follows:

```
DB  1
DB  2
DB  3
```

## Related information

**DUP** (Duplicate Sequence of Source Lines),
**DUPA** (Duplicate Sequence with Arguments),
**DUPF** (Duplicate Sequence in Loop),
**MACRO** (Define Macro)

# DUPF / ENDM

## Syntax

[*label*] **DUPF** *formal_arg*,[*start*],*end*[,*increment*]
.
.
**ENDM**

## Description

With the `DUPF` and `ENDM` directives (Duplicate Sequence in Loop) you can repeat a block of source statements (*end* – *start*) + 1 / *increment* times. *Start* is the starting value for the loop index; *end* represents the final value. *Increment* is the increment for the loop index; it defaults to 1 if omitted (as does the *start* value). The *formal_arg* parameter holds the loop index value and may be used within the body of instructions.

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

## Example

Consider the following source input statements,

```
DUPF    NUM,0,3
MOV.W   R0,NUM
ENDM
```

This is expanded as follows:

```
MOV.W   R0,0
MOV.W   R0,1
MOV.W   R0,2
MOV.W   R0,3
```

## Related information

**DUP** (Duplicate Sequence of Source Lines),
**DUPA** (Duplicate Sequence with Arguments),
**DUPC** (Duplicate Sequence with Characters),
**MACRO** (Define Macro)

# END

## Syntax

**END**

## Description

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

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

A label is not allowed before this directive.

## Example

```
END             ;End of source program
```

## Related information

–

# EQU

## Syntax

*symbol* **EQU** *expression*

## Description

With the `EQU` directive you assign the value of *expression* to *symbol* permanently. Once defined, you cannot redefine the *symbol*.

The *expression* can be relocatable or absolute and forward references are allowed.

## Example

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

```
A_D_PORT  EQU  0x4000
```

You cannot redefine the symbol `A_D_PORT` after this.

## Related information

**SET** (Set temporary value to a symbol)

# EXITM

## Syntax

**EXITM**

## Description

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

A label is not allowed before this directive.

## Example

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

## Related information

**DUP** (Duplicate Sequence of Source Lines),
**DUPA** (Duplicate Sequence with Arguments),
**DUPC** (Duplicate Sequence with Characters),
**DUPF** (Duplicate Sequence in Loop),
**MACRO** (Define Macro)

# EXTERN

## Syntax

**EXTERN** [**(***section_type***)**] *symbol*[**,***symbol*]...

## Description

With the `EXTERN` directive (External Symbol Declaration) you specify that the list of symbols is referenced in the current module, but is not defined within the current module. These symbols must either have been defined outside of any module or declared as globally accessible within another module with the `GLOBAL` directive. The optional argument *section_type* is used for type checking. You can specify the same sections types as with the DEFSECT directive.

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

A label is not allowed before this directive.

## Example

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

EXTERN (code) EE      ;defined elsewhere, of type code
```

## Related information

**GLOBAL** (Global symbol declaration)
**LOCAL** (Local symbol declaration)
**DEFSECT** (Declare a section with name, type and attributes)

# FAIL

## Syntax

**FAIL** [{*string* | *expr*}[**,**{*string* | *expr*}]...]

## Description

With the `FAIL` directive (Programmer Generated Error) you tell the assembler to output an error message during the assembly process.

The total error count will be incremented as with any other error. The `FAIL` directive is for example useful in combination with conditional assembly for exceptional condition checking. The assembly process proceeds normally after the error has been printed.

Optionally, you can specify an arbitrary number of strings and expressions, in any order but separated by commas, to describe the nature of the generated error. If you use expressions, the assembler outputs the result. The assembler outputs a space between each argument.

With this directive the assembler exits with exit code 1 (an error).

A label is not allowed before this directive.

## Example

```
FAIL  'Parameter out of range'
```

This results in the error:

```
E143: ["filename" line] Parameter out of range
```

## Related information

**MSG** (Programmer Generated Message),
**WARN** (Programmer Generated Warning)

# FLOAT/DOUBLE

## Syntax

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

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

## Description

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

An *expression* can be:

- a floating–point expression
- NULL (indicated by two adjacent commas: ,,)

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

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

If the evaluated argument is too large to be represented in 32 / 64 bits, the assembler issues an error and truncates the value.

## Examples

```
FLT:  FLOAT  12.457,+0.27E–13

DBL:  DOUBLE  12.457,+0.27E–13
```

## Related information

**BS** (Block storage)

**DS** (Define storage)

# GLOBAL

## Syntax

**GLOBAL** *symbol*[**,***symbol*]...

## Description

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

With the `GLOBAL` directive (Global Section Symbol Declaration) you declare one of more symbols as global. This means that the specified symbols are defined within the current section or module, and that those definitions should be accessible by all modules, using the `EXTERN` directive.

Only symbols that are defined with the `EQU` or with the `BTEQU` directive or program labels can be made global.

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

A label is not allowed before this directive.

## Example

```
        DEFSECT  "data_io",DATA, init
        SECT     "data_io"
        GLOBAL LOOPA    ; LOOPA will be globally
                        ; accessible by other modules
LOOPA   DW     0x100    ; assigns the value 0x100 to LOOPA
```

## Related information

**EXTERN** (External symbol declaration)
**LOCAL** (Local symbol declaration)

# 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.

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

A label is not allowed before 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   equ   0
DEMO   equ   1
```

You can also define the symbols on the command line with the option **–D**:

```
asm16c –DDEMO –DTEST=0 test.src
```

### Related information

–

# INCLUDE

## Syntax

**INCLUDE** '*filename*' | **<***filename***>**

## Description

With the `INCLUDE` directive you include another file at the exact location in the source where the `INCLUDE` occurs. 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 *filename* specifies the filename of the file to be included. The filename must be compatible with the operating system (forward/backward slashes) and can include a directory specification.

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

1. The current directory if you used 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 **–I**.

3. The path that is specified in the environment variable ASM16CINC when the product was installed.

4. The `include` directory relative to the installation directory.

A label is not allowed before this directive.

## Example

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

## Related information

Assembler option **–I** (Add directory to include file search path) in section 4.2, *Assembler Options*, of Chapter *Tool Options*.

# LOCAL

## Syntax

**LOCAL** *symbol*[,*symbol*]...

## Description

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

With the `LOCAL` directive (Local Section Symbol Declaration) you declare one of more symbols as local. This means that the specified symbols are explicitly local to the module in which you define them.

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

A label is not allowed before this directive.

## Example

```
        DEFSECT  "data_io",DATA
        SECT     "data_io"
        LOCAL  LOOPA   ; LOOPA is local to this section

LOOPA  WORD  0x100   ; assigns the value 0x100 to LOOPA
```

## Related information

**EXTERN** (External symbol declaration)
**GLOBAL** (Global symbol declaration)

# 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 `ENDM` directive indicates the end of the macro.

The definition of a macro consists of three parts:

- *Header*, which assigns a name to the macro and defines the arguments.
- *Body*, which contains the code or instructions to be inserted when the macro is called.
- *Terminator*, which indicates the end of the macro definition (`ENDM` directive).

The arguments are symbolic names that the macro preprocessor 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 (**%**).

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 | Causes local labels in its term to be evaluated at normal scope rather than at macro scope. |

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

### Example

The macro definition:

```
SWAP_REGS  MACRO REG1,REG2                    ;header
     XCHG.W      REG1,REG2                    ;body
     ENDM                                     ;terminator
```

The macro call:

```
DEFSECT  ”code”,code
SECT   ”code”

SWAP_REGS R0,R1
```

The macro expands as follows:

```
XCHG.W  R0,R1
```

### Related information

**DUP** (Duplicate Sequence of Source Lines),
**DUPA** (Duplicate Sequence with Arguments),
**DUPC** (Duplicate Sequence with Characters),
**DUPF** (Duplicate Sequence in Loop)

Section 4.10, *Macro Operations*, in Chapter *Assembly Language* of the *User’s Guide*.

# MSG

## Syntax

**MSG** [{*string* | *expr*}[,{*string* | *expr*}]...]

## Description

With the `MSG` directive (Programmer Generated Message) you tell the assembler to output an information message durring the assembly process.

The error and warning counts will not be affected. The `MSG` directive is for example useful in combination with conditional assembly for informational purposes. The assembly proceeds normally after the message has been printed.

Optionally, you can specify an arbitrary number of strings and expressions, in any order but separated by commas, to describe the nature of the message. If you use expressions, the assembler outputs the result. The assembler outputs a space between each argument.

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

A label is not allowed before this directive.

## Example

```
DEFINE LONG "SHORT"
MSG 'This is a LONG string'
MSG "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
```

## Related information

**FAIL** (Programmer Generated Error)
**WARN** (Programmer Generated Warning)

# PMACRO

## Syntax

**PMACRO** *symbol*[**,***symbol*]...

## Description

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

A label is not allowed before this directive.

## Example

```
PMACRO   MAC1,MAC2
```

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

## Related information

**MACRO** (Define Macro)

# RADIX

## Syntax

**RADIX** *expression*

## Description

With the `RADIX` directive (Change Input Radix for Constants) you tell the assembler to change the input base of constants to the result of *expression*.

The absolute expression must evaluate to one of the legal constant bases (2, 8, 10, or 16). The default radix is 10. The `RADIX` directive allows the programmer to specify constants in a preferred radix without a leading radix indicator. Note that if a constant is used to alter the radix, it must be in the appropriate input base at the time the `RADIX` directive is encountered.

A label is not allowed before this directive.

## Example

```
_RAD10:   DB   10     ; Evaluates to hex A
     RADIX 2
_RAD2:    DB   10     ; Evaluates to hex 2
     RADIX 0x10
_RAD16:   DB   10     ; Evaluates to hex 10
     RADIX 3          ; Bad radix expression
```

## Related information


–

# SECT

### Syntax

**SECT** "*name*" [**, RESET**]

### Description:

With the `SECT` directive you activate a previously declared section with the section name *name*. Before you can activate a section, you must define the section with the `DEFSECT` directive. You can activate a section as many times as you need.

With the section attribute `RESET` you can reset counting storage allocation in `data` sections that have section attribute `max`.

A label is not allowed before this directive.

### Examples:

```
DEFSECT  "text", DATA        ;declare section text
SECT     "text"              ;switch to section text
```

**DEFSECT** (Declare a section with name, type and attributes)

Section 4.9 *Working with Sections* in chapter *Assembly Language* of the *User's Guide*.

# SET

## Syntax

*symbol* **SET** *expression*

**SET** *symbol expression*

## Description

With the `SET` directive you assign the value of *expression* to *symbol*. If a symbol was defined with the `SET` directive, you can redefine that symbol in another part of the assembly source, using the `SET`.

The `SET` directive is useful in establishing temporary or reusable counters within macros. *Expression* must be absolute and forward references are not allowed.

Symbols that are set with the EQU directive, cannot be redefined.

## Example

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

## Related information

**EQU** (Assign permanent value to a symbol)

# SIZE

## Syntax

**SIZE** *symbol*, *expression*

## Description

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

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

## Example

```
_main:  type func
   .      ; function _main
   .
   rts
   SIZE _main,($-_main)
```

## Related information

**TYPE** (Set Symbol Type)

# TYPE

## Syntax

*symbol* **TYPE** *typeid*

## Description

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

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

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

## Example

```
Afunc:   TYPE   FUNC
```

## Related information

**SIZE** (Set Symbol Size)

# UNDEF

## Syntax

**UNDEF** *symbol*

## Description

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

A label is not allowed before this directive.

## Example

```
UNDEF  LEN    ; Undefines the LEN substitution string
              ; that was previously defined with the
              ; DEFINE directive
```

## Related information

**DEFINE** (Define Substitution String)

# WARN

## Syntax

**WARN** [{*string* | *expr*}[,{*string* | *expr*}]...]

## Description

With the `WARN` directive (Programmer Generated Warning) you tell the assembler to output a warning message during the assembly process.

The total warning count will be incremented as with any other warning. The `WARN` directive is for example useful in combination with conditional assembly for exceptional condition checking. The assembly process proceeds normally after the warning has been printed.

Optionally, you can specify an arbitrary number of strings and expressions, in any order but separated by commas, to describe the nature of the generated warning. If you use expressions, the assembler outputs the result. The assembler outputs a space between each argument.

This directive has no effect on the exit code of the assembler, unless you use the assembler option **--warnings-as-errors**. In that case the assembler exits with exit code 1 (an error).

A label is not allowed before this directive.

## Example

```
WARN  'parameter too large'
```

This results in the warning:

```
W144: ["filename" line] Parameter out of range
```

## Related information

**FAIL** (Programmer Generated Error),
**MSG** (Programmer Generated Message)

# WEAK

## Syntax

**WEAK** *symbol*[,*symbol*]...

## Description

With the `WEAK` directive you mark one of 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 LOOPA as weak
```

## Related information


–

### 3.3.3 OVERVIEW OF ASSEMBLER CONTROLS

The following tables provide an overview of all assembler controls. Note that most of them have an equivalent assembler command line option.

***Overview of assembler listing controls***

| Function | Description |
|---|---|
| $LIST ON / OFF | Generation of assembly list file temporary ON/OFF |
| $LIST "*flags*" | Exclude / include lines in assembly list file |
| $PAGE | Generate formfeed in assembly list file |
| $PAGE *settings* | Define page layout for assemly list file |
| $PRCTL | Send control string to printer |
| $STITLE *string* | Set program subtitle in header of assembly list file |
| $TITLE *string* | Set program title in header of assembly list file |

***Overview of miscellaneous assembler controls***

| Function | Description |
|---|---|
| $CASE ON / OFF | Case sensitive user names ON/OFF |
| $DEBUG ON / OFF | Generation of symbolic debug ON/OFF |
| $DEBUG "*flags*" | Select debug information |
| $IDENT LOCAL / GLOBAL | Assembler treats labels by default as local or global |
| $OBJECT | Alternative name for the generated object file |
| $OPTJ | Turn on conditional optimization |
| $WARNING OFF [*num*] | Suppress all or some warnings |

### 3.3.4 DETAILED DESCRIPTION OF ASSEMBLER CONTROLS

The assembler recognizes both upper and lower case for controls.

# $CASE ON / OFF

## Syntax

**$CASE ON** (default)
**$CASE OFF**

## Description

With the `$CASE ON` and `$CASE OFF` controls you specify whether the assembler operates in case sensitive mode or not. By default the assembler operates in case sensitive mode. This means that all user–defined symbols and labels are treated case sensitive, so `LAB` and `Lab` are distinct. Note that instruction mnemonics, register names, directives and controls are always treated case insensitive.

## Example

```
;begin of source
$CASE OFF   ; assembler in case insensitive mode
```

## Related option

Assembler option **–c** (Switch to case insensitive mode) in section 4.2, *Assembler Options*, of Chapter *Tool Options*.

## Related information

–

# $DEBUG ON / OFF

## Syntax

**$DEBUG ON**
**$DEBUG OFF**
**$DEBUG** "*flags*"

## Description

With the `$DEBUG ON` and `$DEBUG OFF` controls you turn the generation of debug infomation on or off. (`$DEBUG ON` is similar to the assembler option **–gl**).

If you use `$DEBUG` control with flags, you can set the following flags:

**a**/**A** assembler source line information
**h**/**H** pass HLL debug information

You cannot use these two types of debug information both. So, `$DEBUG "ah"` is not allowed.

**l**/**L** local symbols debug information
**s**/**S** always debug; either "**AhL**" or "**aHl**"

Debug information that is generated by the C compiler, is *always* passed to the object file.

## Example

```
;begin of source
$DEBUG ON   ; generate local symbols debug information
```

## Related option

Assembler option **–g** (Select debug information) in section 4.2, *Assembler Options*, of Chapter *Tool Options*.

## Related information

–

# $IDENT

## Syntax

**$IDENT LOCAL**
**$IDENT GLOBAL**

## Description

With the controls `$IDENT LOCAL` and `$IDENT GLOBAL` you tell the assembler how to treat symbols that you have not specified explicitly as local or global with the assembler directives `LOCAL` or `GLOBAL`.

Default the assembler treats all symbols as local symbols unless you have defined them explicitly as global.

## Example

```
;begin of source
$IDENT GLOBAL  ; assembly labels are global by default
```

## Related option

Assembler option **–i** (Treat labels by default local / global) in section 4.2, *Assembler Options*, of Chapter *Tool Options*.

## Related information

Assembler directive **LOCAL** (Local symbol declaration)
Assembler directive **GLOBAL** (Global symbol declaration)

# $LIST ON / OFF

## Syntax

**$LIST ON**

.
. ; assembly source lines
.

**$LIST OFF**

## Description

If you generate a list file with the assembler option **–l**, you can use the `$LIST ON` and `$LIST OFF` controls to specify which source lines the assembler must write to the list file. Without the command line option **–l**, the `$LIST ON` and `$LIST OFF` controls have no effect.

The `$LIST ON` control actually increments a counter that is checked for a positive value and is symmetrical with respect to the `$LIST OFF` control. Note the following sequence:

```
; Counter value currently 1
$LIST ON          ; Counter value = 2
$LIST ON          ; Counter value = 3
$NOLIST OFF       ; Counter value = 2
$NOLIST OFF       ; Counter value = 1
```

The listing still would not be disabled until another `NOLIST` control was issued.

## Example

Suppose you assemble the following assembly source with the assembler option **–l**:

```
DEFSECT "text_CO",CODE
SECT  "text_CO"
...  ; source line in list file
$LIST ON
...  ; source line not in list file
$LIST
...  ; source line also in list file
END
```

The assembler generates a list file with the following lines:

```
DEFSECT "text_CO",CODE
SECT  "text_CO"
...  ; source line in list file
$LIST ON
...  ; source line also in list file
END
```

### Related option

Assembler option **–l** (Generate list file) in section 4.2, *Assembler Options*, of Chapter *Tool Options*.

### Related information

Assembler control **$LIST** (Exclude / include lines in assembly list file)

Assembler function **@LST()** in section 3.2, *Built–in Asembly Functions*.

# $LIST *flags*

## Syntax

*Begin of assembly file*

**$LIST "***flags***"**

## Description

If you generate a list file with the assembler option **–l**, you can use the `$LIST` controls to specify which type of source lines the assembler must exclude from the list file. Without the command line option **–l**, the `$LIST` control has no effect.

You can set the following flags to remove or include lines:

**c**/**C** Lines with assembler controls
**d**/**D** Lines with section directives (SECT and DEFSECT)
**e**/**E** Lines with symbol definition directives (EXTERN, GLOBAL, LOCAL, CALLS)
**g**/**G** Lines with generic instruction expansion
**i**/**I** Lines with generic instructions
**m**/**M** Lines with macro definitions (MACRO and DUP)
**n**/**N** Empty source lines
**p**/**P** Lines with conditional assembly
**q**/**Q** Lines with the EQU or SET directive
**r**/**R** Relocation characters ('r')
**s**/**S** Lines with symbolic debug information
**v**/**V** Assembler EQU or SET values
**w**/**W** Wrapped part of a line
**x**/**X** Lines with expanded macros
**y**/**Y** Lines with cycle counts

If you do not specify this control or the assembler option **–L***flag*, the assembler uses the default: **–LcDEGilMnPqrsVWXy**.

## Example

To exclude assembly files with controls from the list file:

```
;begin of source
$LIST "C"
```

### Related option

Assembler option **–L** (List file formatting options) in section 4.2, *Assembler Options*, of Chapter *Tool Options*.

### Related information

Assembler control **$LIST ON / OFF** (Assembly list file ON / OFF)

Assembler function **@LST()** in section 3.2, *Built–in Asembly Functions*.

# $OBJECT

## Syntax

**$OBJECT "***file***"**
**$OBJECT OFF**

## Description

With the `$OBJECT` control you can specify an alternative name for the generated object file. With the `$OBJECT OFF` control, the assembler does not generate an object file at all.

## Example

```
;Begin of source
$object "x1.obj"          ; generate object file x1.obj
```

## Related option

Assembler option **–o** (Define output filename) in section 4.2, *Assembler Options*, of Chapter *Tool Options*.

## Related information

–

# $OPTJ

## Syntax

**$OPTJ** [**on** | **off**]

## Description

With the `$OPTJ` control you can turn on or off conditional jump optimization. This control overrules the **–O** command line option.

## Example

To enable jump and branch optimization, enter:

```
$OPTJ ON
```

## Related option

Assembler option **–Oj** (Assembler optimizations) in Section 4.2, *Assembler options*, of Chapter *Tool Options*.

## Related information

–

# $PAGE

## Syntax

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

## Description

If you generate a list file with the assembler option **–l**, you can use the `$PAGE` control to format the generated list file.

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

*length* Number of lines per page (10–255). Default is 66. As a special case a page length of 0 (zero) turns off all headers, titles, subtitles, and page breaks.

*blanktop* Number of blank lines at the top of the page. Default = 0. Specify a value so that *blanktop* + *blankbtm* ≤ *length* – 10.

*blankbtm* Number of blank lines at the bottom of the page. Default = 0. Specify a value so that *blanktop* + *blankbtm* ≤ *length* – 10.

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

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

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

## Example

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

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

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

## Related option


–

### Related information

Assembler control **$STITLE** (Set program subtitle in header of list file)
Assembler control **$TITLE** (Set program title in header of list file)

Assembler option **–l** (Generate list file) in Section 4.2, *Assembler Options*, of Chapter *Tool Options*.

Assembler option **–L** (List file formatting options) in Section 4.2, *Assembler Options*, of Chapter *Tool Options*.

# $PRCTL

## Syntax

**$PRCTL** *exp*|*string*[,*exp*|*string*]...

## Description

If you generate a list file with the assembler option **–l**, you can use the `$PRCTL` control to send control strings to the printer.

The `$PRCTL` control simply concatenates its arguments and sends them to the listing file (the control line itself is not printed unless there is an error).

You can specify the following arguments:

*exp* — a byte expression which may be used to encode non–printing control characters, such as ESC.

*string* — an assembler string. which may be of arbitrary length, up to the maximum assembler–defined limits.

The `$PRCTL` control can appear anywhere in the source file; the assembler sends out the control string at the corresponding place in the listing file.

If a `$PRCTL` control is the last line in the last input file to be processed, the assembler insures that all error summaries, symbol tables, and cross–references have been printed before sending out the control string. In this manner, you can use a `PRCTL` control to restore a printer to a previous mode after printing is done.

Similarly, if the `$PRCTL` control appears as the first line in the first input file, the assembler sends out the control string before page headings or titles.

## Example

```
$PRCTL  $1B,'E'  ; Reset HP LaserJet printer
```

## Related option

–

## Related information

Assembler option **–l** (Generate list file) in Section 4.2, *Assembler Options*, of Chapter *Tool Options*.

# $STITLE

## Syntax

**$STITLE** "*title*"

## Description

If you generate a list file with the assembler option **–l**, you can use the `$STITLE` control to specify the program subtitle which is printed at the top of all succeeding pages in the assembler list file below the title.

The specified subtitle is valid until the assembler encouters a new `STITLE` control. Default, the subtitle is empty.

The `$STITLE` control itself will not be printed in the source listing.

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

## Example

```
$TITLE   'This is the title'
$STITLE  'This is the subtitle'
```

The header of the second page in the list file will now be:

```
TASKING M16C Assembler vx.yrz Build nnn SN 00000000
This is the title                                    Page 2
This is the subtitle
```

## Related option

–

## Related information

Assembler control **$TITLE** (Set program title in header of list file)

Assembler option **–l** (Generate list file) in Section 4.2, *Assembler Options*, of Chapter *Tool Options*.

# $TITLE

## Syntax

**$TITLE** "*title*"

## Description

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

Default, the title is empty.

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

## Example

```
$TITLE  'This is the title'
```

The header of the list file will now be:

```
TASKING M16C Assembler vx.yrz Build nnn SN 00000000
This is the title                                    Page 1
```

## Related option

–

## Related information

**STITLE** (Set program subtitle in header of assembly list file)

# $WARNING OFF

## Syntax

**$WARNING OFF**
**$WARNING OFF** *number*

## Description

With the $WARNING OFF control you can suppresses all warning messages or specific warning messages.

- Default, 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.

## Example

```
$WARNING OFF     ; all warning messages are suppressed

$WARNING OFF 135 ; suppress warning message 135
```

## Related option

Assembler option **–w** (Suppress some or all warnings) in section 4.2, *Assembler Options*, of Chapter *Tool Options*.

## Related information

–

CHAPTER 4

# TOOL OPTIONS

TASKING

# 4 CHAPTER

## 4.1 COMPILER OPTIONS

This section lists all compiler options.

### *Options in EDE versus options on the command line*

Most command line options have an equivalent option in EDE but some options are only available on the command line. If there is no equivalent option in EDE, you can specify a command line option in EDE as follows:

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **C Compiler** entry and select **Miscellaneous**.

3. Enter one or more command line options in the **Additional compiler options** field.

Be aware that some command line options are not useful in EDE 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 EDE.

### *Short and long option names*

Options 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:

```
cm16c –Oac test.c
cm16c ––optimize=+coalesce,+cse test.c
```

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

# –? (––help)

## EDE

–

## Command line syntax

**–?**

**––help**[=*item*]

You can specify the following arguments:

| | |
|---|---|
| **intrinsics** | Show the list of intrinsic functions |
| **options** | Show extended option descriptions |
| **pragmas** | Show the list of supported pragmas |

## Description

Displays an overview of all command line options. When you specify an argument you can list extended information such as a list of intrinsic functions, pragmas or option descriptions.

## Example

The following invocations all display a list of the available command line options:

```
cm16c –?
cm16c ––help
cm16c
```

The following invocation displays a list of the available pragmas:

```
cm16c ––help=pragmas
```

## Related information

–

# –A (––language)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **C Compiler** entry and select **Language**.

3. Enable or disable the options **Allow C++ style comments in ISO C90 mode and Relax const check for string literals**.

### Command line syntax

**–A**[*flags*]

**––language=**[*flags*]

You can set the following flags:

| | | |
|---|---|---|
| **p**/**P** | (**+**/**–comments**) | Allow C++ style comments in ISO C90 |
| **x**/**X** | (**+**/**–strings**) | Relaxed const check for string literals |

### Default

**–Apx**

### Description

With this option you control the language extensions the compiler can accept. Default the C compiler allows all language extensions.

**–A** (**––language**) is the equivalent of **–APX** and disables all language extensions.

With **–Ap** you tell the compiler to allow C++ style comments (`//`) in ISO C90 mode (option **–c90**). In ISO C99 mode this style of comments is always accepted.

With **–Ax** you tell the compiler not to check for assignments of a constant string to a non–constant string pointer. With this option the following example does not produces a warning:

```
char *p;
void main( void ) { p = ”hello”; }
```

### Example

```
cm16c –APx –c90 test.c
cm16c --language=-comments,+strings --iso=90 test.c
```

The compiler compiles in ISO C90 mode, accepts assignments of a constant string to a non–constant string pointer but ignores C++ style comments.

### Related information

Compiler option **–c** (ISO C standard)

# –-align

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **C Compiler** entry and select **Alignment**.

3. Enable the option **Align all objects to an even address**.

### Command line syntax

**–-align**

### Description

By default the **cm16c** compiler aligns objects to the minimum alignment required by the architecture. With this option you force the compiler to align all objects greater than 8 bits to an even address. This optimizes access time for a 16–bit address bus to functions but may take extra memory space.

### Example

To align all objects to even addresses, enter:

```
cm16c --align test.c
```

### Related information

Compiler option **–-align–data** (Align data to an even address)
Compiler option **–-align–func** (Align functions to an even address)

# –-align-data

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **C Compiler** entry and select **Alignment**.

3. Enable the option **Align data to an even address**.

### Command line syntax

**–-align–data**

### Description

By default the **cm16c** compiler aligns objects to the minimum alignment required by the architecture. With this option you force the compiler to align 16, 32 and 64 bit data variables to even addresses. This optimizes access time for a 16–bit address bus but may take extra memory space.

### Example

To align all data to even addresses, enter:

```
cm16c –-align–data test.c
```

### Related information

Compiler option **–-align** (Align everything to an even address)
Compiler option **–-align–func** (Align functions to an even address)

# –align-func

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **C Compiler** entry and select **Alignment**.

3. Enable the option **Align functions to an even address**.

### Command line syntax

**–align–func**

### Description

By default the **cm16c** compiler aligns objects to the minimum alignment required by the architecture. With this option you force the compiler to align all functions to an even address. This optimizes access time for a 16–bit address bus to functions but may take extra memory space.

### Example

To align all functions to even addresses, enter:

```
cm16c –align–func test.c
```

### Related information

Compiler option **–align** (Align everything to an even address)
Compiler option **–align–data** (Align data to an even address)

# –C (––cpu)

## EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Processor** entry and select **Processor Definition**.

3. Select a processor from the **Select processor** box.

## Command line syntax

**–C***cpu*

**––cpu=***cpu*

## Description

With this option you define the target processor for which you create your application.

Based on the target processor, the compiler includes a *special function register file* `regcpu.sfr`. This is an include file written in C syntax which is shared by the compiler, assembler and debugger. Once the compiler reads an SFR file you can reference the special function registers (SFR) and bits within an SFR using symbols defined in the SFR file.

When you select an R8C target, you always must use the compiler option **––r8c** as well !

## Example

To compile the file `test.c` for the m30624 processor and use the SFR file `regm30624a.sfr`:

```
cm16c –Cm30624a test.c
cm16c ––cpu=m30624a test.c
```

To avoid conflicts, make sure you specify the same target processor to the assembler.

### Related information

Assembler option **–C** (Select CPU)
Control program option **–C** (Select target CPU)

Compiler option **––r8c** (Compile for R8C/tiny)

# –c (––iso)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **C Compiler** entry and select **Language**.

3. Select the ISO C standard **C90** or **C99**.

### Command line syntax

**–c**{**90**|**99**}

**––iso=**{**90**|**99**}

### Default

**–c99**

### 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:

```
cm16c –c90 test.c
cm16c ––iso=90 test.c
```

### Related information

Compiler option **–A** (Language extensions)

# --check

### EDE

1. In the project window, select the file you want to check.
2. From the **Build** menu, select **Check Syntax**.

### Command line syntax

**--check**

### Description

With this option you can check the source code for syntax errors, without generating code. This saves time in developing your application.

The compiler reports any warnings and/or errors.

### Example

To check for syntax errors, without generating code:

```
cm16c --check test.c
```

### Related information

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

# –-compact-max-size

## EDE

–

## Command line syntax

**–-compact–max–size=***value*

## Default

**–-compact–max–size=200**

## Description

This option is related to the compiler optimization **–Or** (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 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:

```
cm16c –Or –-compact–max–size=100 test.c
cm16c –-optimize=+compact –-compact–max–size=100 test.c
```

## Related information

Compiler option **–-max–call–depth** (Maximum call depth for code compaction)
Compiler option **–Or** (Optimization: code compaction)

# –D (––define)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **C Compiler** entry and select **Preprocessing**.

3. Click on an empty **Macro** field and enter a macro name.

4. Optionally, click in the **Definition** field and enter a definition.

### Command line syntax

**–D***macro_name*[**=***macro_definition*]

**––define=***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. On the command line, use the option **–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 **–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:

```
cm16c -DDEMO test.c
cm16c -DDEMO=1 test.c

cm16c --define=DEMO test.c
cm16c --define=DEMO=1 test.c
```

Note that all four 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.

```
cm16c -D"MAX(A,B)=((A) > (B) ? (A) : (B))"
```

### Related information

Compiler option **-U** (Remove preprocessor macro)
Compiler option **-f** (Read options from file)

# --diag

### EDE

1. In the **Help** menu, enable the option **Show Help on Tool Errors**.
2. In the **Build** tab of the **Output** window, double–click on an error or warning message.

   *A description of the selected message appears.*

### Command line syntax

**--diag=**[*format***:**]{**all** | *nr*[**,***nr*]...}

Optionally, you can use one of the following display formats (*format*):

| | |
|---|---|
| **text** | The default is plain text |
| **html** | Display explanation in HTML format |
| **rtf** | Display explanation in RTF format |

### 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.

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 compiler does not compile any files.

### Example

To display an explanation of message number 282, enter:

```
cm16c --diag=282
```

This results in the following message and explanation:

```
E282: unterminated comment

Make sure that all 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 a file named `cerrors.html`, enter:

```
cm16c --diag=html:all > cerrors.html
```

### Related information

–

# –E (––preprocess)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **C Compiler** entry and select **Preprocessing**.

3. Enable the option **Store the C compiler preprocess output (<file>.pre)**.

### Command line syntax

**–E**[*flags*]

**––preprocess**[**=***flags*]

You can set the following flags (when you specify **–E** without flags, the default is **–ECMP**):

| | | |
|---|---|---|
| **c**/**C** | (**+**/**–comments**) | Keep comments |
| **m**/**M** | (**+**/**–make**) | Generate dependencies for make |
| **p**/**P** | (**+**/**–noline**) | Strip #line source position info |

### Description

With this option you tell the compiler to preprocess the C source. EDE stores the preprocess output in the file ***name*`.pre`** (where *name* is the name of the C source file to compile). EDE 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 **–o**.

With **–Ec** you tell the preprocessor to keep the comments from the C source file in the preprocessed output.

With **–Em** the compiler will generate dependency lines that can be used in a Makefile. The preprocessor output is discarded.

With **–Ep** 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

```
cm16c –EcMP test.c –o test.pre

cm16c --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


–

# --error-file

### EDE

–

### 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:

```
cm16c --error-file=errors.err test.c
```

### Related information

Compiler option **--warnings-as-errors** (Treat warnings as errors)

# –F (––no-double)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **C Compiler** entry and select **Floating Point**.

3. Enable the option **Use single precision floating point only**.

### Command line syntax

**–F**

**––no–double**

### 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

```
cm16c –F test.c
cm16c ––no–double test.c
```

The file `test.c` is compiled where variables of the type `double` are treated as `float`.

### Related information

–

# –f (––option-file)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **C Compiler** entry and select **Miscellaneous**.

3. Add the option **–f** to the **Additional compiler options** field.

In EDE you can save your options in a file and restore them to call the compiler with those options:

- From the **Project** menu, select **Save Options...** or **Load Options...**

Be aware that when you specify the option **–f** in the **Additional compiler options** field, the options are *added* to the compiler options you have set in the Project Options dialog. Only in extraordinary cases you may want to use them in combination.

### Command line syntax

**–f** *file*

**––option–file=***file*

### Description

Instead of typing all options on the command line, you can create an option file which contains all options and files you want to specify. With this option you specify the option file to the compiler.

Use an option file when the length of the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option **–f** multiple times.

#### *Format of an option file*

- Multiple command line 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:

```
-Cm30624
-s
test.c
```

Specify the option file to the compiler:

```
cm16c -f myoptions
cm16c --option-file=myoptions
```

This is equivalent to the following command line:

```
cm16c -Cm30624 -s test.c
```

### Related information

–

# –g (––debug-info)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **C Compiler** entry and select **Debug**.

3. Enable the option **Generate symbolic debug infomation**

### Command line syntax

**–g**

**––debug–info**

### 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 code size. For the final application, compile your C files without debug information.

When you specify a high optimization level, the debug comfort may decrease. Therefore, the compiler issues warning W555 if the debug comfort would be decreased as a result of the chosen optimizations.

### Example

To add symbolic debug information to the output file, enter:

```
cm16c –g test.c
cm16c ––debug test.c
```

### Related information


–

# –H (––include-file)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **C Compiler** entry and select **Preprocessing**.

3. Enter the name of the file in the **Include this file before source** field.

### Command line syntax

**–H***file*,...

**––include–file=***file*,...

### Description

With this option you include one extra file 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

```
cm16c –Hstdio.h test1.c test2.c
cm16c ––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

Compiler option **–I** (Add directory to include file search path)

Section 5.5, *How the Compiler Searches Include Files*, in Chapter *Using the Compiler* of the *User's Guide*.

# –I (––include-directory)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Open the **Build Options** tab.

3. Enter one or more search paths in the **Include Files Path** field.

   *If you enter multiple paths, separate them with a semicolon (;).*

### Command line syntax

**–I***path***,**...

**––include–directory=***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 CM16CINC when the product was installed.
4. The default `include` 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:

```
cm16c –Imyinclude test.c
cm16c ––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

Compiler option **–H** (Include file at the start of a compilation)

Section 5.5, *How the Compiler Searches Include Files*, in Chapter *Using the Compiler* of the *User's Guide*.

Section 1.3.2, *Configuring the Command Line Environment*, in Chapter *Software Installation* of the *User's Guide*.

# –-inline

### EDE

–

### Command line syntax

**–-inline**

### Description

With this option you instruct the compiler to inline *all* functions, regardless whether they have the keyword `inline` or not. 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 (option **–Or**).

### Example

To inline all functions:

```
cm16c --inline test.c
```

### Related information

Compiler option **–Or** (Optimization: code compaction)

# –-inline-max-incr / –-inline-max-size

### EDE

–

### Command line syntax

**–-inline–max–incr=***percentage*

**–-inline–max–size=***threshold*

### Default

**–-inline–max–incr=25**

**–-inline–max–size=10**

### Description

With these options you can control the function inlining optimization process of the compiler. These options have only effect when you have enabled the inlining optimization (option **–Oi**).

Regardless of the optimization process, the compiler always inlines *all* functions that have the function qualifier `inline`.

With the option **–-inline–max–size** you can specify the maximum size of functions that the compiler inlines as part of the optimization process. The compiler always inlines *all* functions that are smaller than the specified *threshold*. The threshold is measured in compiler internal units and the compiler uses this measure to decide which functions are small enough to inline. The default threshold is 10.

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. Default, this is 25% which means that the compiler continues inlining functions until the resulting code size is 25% larger than the original size.

### Example

```
cm16c --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

Compiler option **–O** (Specify optimization level)

Section 3.12.3, *Inlining Functions*, in Chapter *C Language* of the *User's Guide*.

# –-integer-enumeration

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **C Compiler** entry and select **Miscellaneous**.

3. Enable the option **Use 16–bit integers for enumeration**.

### Command line syntax

**–-integer–enumeration**

### Description

Normally the compiler treats small enumerated types as `char` or even as `__bit` instead of `int`. This reduces code size. With this option the compiler always treats enum–types as integer.

### Example

To treat enumerated types always as integer, enter:

```
cm16c --integer-enumeration test.c
```

### Related information

–

# –k (––keep-output-files)

## EDE

EDE always removes the `.src` file when errors occur during compilation.

## Command line syntax

**–k**

**––keep–output–files**

## 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 **mkm16c**. 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

```
cm16c –k test.c
```

When an error occurs during compilation, the generated output file `test.src` will *not* be removed.

## Related information

Compiler option **––warnings–as–errors** (Treat warnings as errors)

# –M (––model)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **C Compiler** entry and select **Memory Model**.

3. Choose a memory model.

### Command line syntax

**–M{l | m | s}**

**––model={large | medium | small}**

### Description

By default the **cm16c** compiler uses the small memory model. This model generates the most efficient code. You can use the option **–M** to specify another memory model to the control program.

The table below illustrates the meaning of each data model:

| Model | Data | Constants | Pointers |
|---|---|---|---|
| Small | __near: in first 64 kB | __near | __near |
| Medium | __near: in first 64 kB | __paged | __paged |
| Large | __far: anywhere in 1 MB | __far | __far |

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:

small model 's'
medium model 'm'
large model 'l'

### Example

To compile the file `test.c` for the large memory model:

```
cm16c -Ml test.c
cm16c --model=large test.c
```

### Related information

Section 3.5, *Memory Models*, in Chapter *C Language* of the *User's Guide*.

# –-max-call-depth

### EDE

–

### Command line syntax

**–-max–call–depth=***value*

### Default

**–-max–call–depth=–1**

### Description

This option is related to the compiler optimization **–Or** (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 at the cost of execution speed.

During code compaction 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 **–OR**)

**>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 **–Or** 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.

### Example

To limit the call depth resulting from code compaction to 10, enter:

```
cm16c -Or --max-call-depth=10 test.c
cm16c --optimize=+compact --max-call-depth=10 test.c
```

### Related information

Compiler option **--compact-max-size** (Maximum size of a match for code compaction)
Compiler option **-Or** (Optimization: code compaction)

# –misrac

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **C Compiler** entry and select **MISRA C**.
3. Select a MISRA C configuration.
4. (Optional) In the **MISRA C Rules** entry, specify the individual rules.

### Command line syntax

**–misrac={all** | *number* [**–***number*]**,**... }

### Description

With this option you specify to the compiler which MISRA C rules must be checked. With the option **–misrac=all** the compiler checks for all supported MISRA C rules.

### Example

```
cm16c –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

See Chapter 8 *MISRA C Rules* for a list of all supported MISRA C rules.

Linker option **–misra–c–report**.

# –n (––stdout)

### EDE

–

### Command line syntax

**–n**

**––stdout**

### 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.

### Example

```
cm16c –n test.c
cm16c ––stdout test.c
```

The compiler sends the output (normally `test.src`) to stdout and does not create the file `test.src`.

### Related information

–

# --near-rom

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **C Compiler** entry and select **Code Generation**.

3. Enable the option **ROM is available in first 64k of memory**.

### Command line syntax

**--near-rom**

### Description

Usual M16C hardware configurations have no ROM in near memory (first 64K of memory). So, by default, `__near` qualified objects are allocated in RAM, even with options **Keep strings in ROM** or **Keep constants in ROM** enabled.

If your hardware does have ROM in near memory, you can specify this with compiler option **--near-rom**.

### Example

To keep strings in ROM in near memory:

```
cm16c --near-rom --romstrings test.c
```

### Related information

Compiler options **--romstrings** / **--romconstants**

# –-noclear

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog box appears.*

2. Expand the **C Compiler** entry and select **Miscellaneous**.

3. Add the option **–-noclear** to the **Additional C Compiler options** field.

### Command line syntax

**–-noclear**

### Description

Normally variables are cleared at program startup. With this option you tell the compiler to generate code to prevent non–initialized global variables from being cleared at program startup.

### Example

To prevent non–initialized global variables in the module `test.c` from being cleared at program startup, enter:

```
cm16c --noclear test.c
```

### Related information

–

# --noframe

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog box appears.*

2. Expand the **C Compiler** entry and select **Code Generation**.

3. *Disable* the option **Generate frame for interrupt handler**.

### Command line syntax

**--noframe**

### Description

This option tells the compiler not to generate an interrupt frame (saving/restoring registers) for interrupt handlers.

### Example

To disable the generation of an interrupt frame:

```
cm16c --noframe test.c
```

### Related information

Compiler option **--novector** (do not generate interrupt vectors)

# --novector

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog box appears.*

2. Expand the **C Compiler** entry and select **Code Generation**.

3. *Disable* the option **Generate code for fixed interrupt vector**.

### Command line syntax

**--novector**

### Description

With this option you tell the compiler not to generate code for interrupt vectors and references to the interrupt handler in the run–time library.

### Example

To disable code generation for interrupt vectors:

```
cm16c --novector test.c
```

### Related information

Compiler option **--noframe** (do not generate frame for interrupt handler)

# –O (––optimize)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **C Compiler** entry and select **Optimization**.

3. Select an optimization level in the **Optimization level** box.

### Command line syntax

**–O**[*flags*]

**––optimize**[**=***flags*]

You can set the following flags:

| | | |
|---|---|---|
| **a**/**A** | (**+**/**–coalesce**) | Coalescer: remove unnecessary moves |
| **b**/**B** | (**+**/**–ipro**) | Interprocedural register optimizations |
| **c**/**C** | (**+**/**–cse**) | Common subexpression elimination |
| **e**/**E** | (**+**/**–expression**) | Expression simplification |
| **f**/**F** | (**+**/**–flow**) | Control flow optimization and code reordering |
| **g**/**G** | (**+**/**–glo**) | Generic assembly optimizations |
| **i**/**I** | (**+**/**–inline**) | Function inlining |
| **l**/**L** | (**+**/**–loop**) | Loop transformations |
| **o**/**O** | (**+**/**–forward**) | Forward store |
| **p**/**P** | (**+**/**–propagate**) | Constant propagation |
| **r**/**R** | (**+**/**–compact**) | Code compaction (reverse inlining) |
| **s**/**S** | (**+**/**–subscript**) | Subscript strength reduction |
| **y**/**Y** | (**+**/**–peephole**) | Peephole optimizations |
| **z**/**Z** | (**+**/**–call–to–jump**) | Replace call; return by jump |

Use the following options for predefined sets of flags:

| | | |
|---|---|---|
| **–O0** | (**––optimize=0**) | No optimization.<br>Alias for: **–OABCEFGILOPRSYZ** |
| **–O1** | (**––optimize=1**) | Debug purpose optimizations<br>Alias for: **–OabcefgILOpRSyZ** |
| **–O2** | (**––optimize=2**) | Release purpose optimization<br>(default, debugging possible)<br>Alias for: **–OabcefgIlopRsyz** |

**–O3** (**––optimize=3**) Full optimization
Alias for: **–Oabcefgiloprsyz**

### Default

**–O2**

### Description

With this option you can control the level of optimization. If you do not use this option, the default optimization level is *release purpose optimization* (option **–O2** or **–O** or **–OabcefgIlopRsyz**).

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`* and `#pragma endoptimize`.

In addition to the option **–O**, you can specify the option **–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 release purpose optimization set:

```
cm16c test.c

cm16c –O2 test.c
cm16c ––optimize=2 test.c

cm16c –O test.c
cm16c ––optimize test.c

cm16c –OabcefgIlopRsy test.c
cm16c ––optimize=+coalesce,+ipro,+cse,+expression,
    +flow,+glo,–inline,+loop,+forward,+propagate,
    –compact,+subscript,+peephole test.c
```

### Related information

Compiler option **–t** (Trade off between speed (**–t0**) and size (**–t4**))

```
#pragma optimize flag
#pragma endoptimize
```

Section 5.3, *Compiler Optimizations*, in Chapter *Using the Compiler* of the *User's Guide*.

# –o (––output)

## EDE

–

## Command line syntax

**–o***file*

**––output=***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`.

EDE names the output file always after the C source file.

## Example

```
cm16c -o output.src test.c
cm16c --output=output.src test.c
```

The compiler creates the file `output.src` for the compiled file `test.c`.

Without the option **–o**, like EDE, the compiler uses the names of the input file and creates `test.src`.

## Related information

–

# –R (––rename-sections)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **C Compiler** entry and select **Miscellaneous**.

3. Add the option **–R** to the **Additional compiler options** field.

### Command line syntax

**–R** [*type*][**=***name*]

**––rename–sections**[**=***type*][**=***name*]

The *type* is a two–letter abbreviation indicating the memory type. You can specify the following memory types:

| Type | Description |
|---|---|
| CO | program code |
| DA | __near data |
| FD | __far data |
| BI | __bit type section |
| BA | __bita type section (bitaddressable data) |

### Description

The compiler uses the following method to create section names: it uses the module name and a two letter memory type abbreviation to create the name *name_type*. For example, if you compile the C source file `test.c`, the compiler creates the name `test_CO` for executable code sections.

In case a module must be loaded at a fixed address or a data section needs a special place in memory, you can use the option **–R** to generate a unique section name. Having unique names, you can use the Linker Script Language (LSL) to control the location of these sections.

When you use **–R** without a value, the compiler uses the default section naming.

### Example

To create a new section name for `__bit` sections, enter:

```
cm16c -RBI=MARK test.c
cm16c --rename-section=BI=MARK test.c
```

Without the option **-R** the bit sections would have received the name `TEST_`*`TYPE`*`_BI`. During renaming, the type is preserved.

### Related information

Section 3.13, *Section Naming*, in Chapter *C Language* of the *User's Guide*.

# --r8c

## EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Processor** entry and select **Processor Definition**.

3. Select a processor from the **Select processor** box.

## Command line syntax

**--r8c**

## Description

By default, the compiler generates code for the M16C/60 core. With this option you tell the compiler to compile for R8C/tiny core. You must use this option always (and only then) when you select an R8C target with the compiler option **-C***cpu*. In EDE this option is automatically enabled when you select an R8C target.

When you compile for the R8C/tiny:

- Other vector addresses are generated with the `__interrupt_fixed` keyword
- `__far` and `__paged` memory type qualifiers are interpreted as `__near`
- because code resides in near memory, pointers to functions are 16 bit (instead of 32 bit)

To avoid conflicts, make sure you specify this option also to the assembler.

## Example

```
cm16c --r8c --Cr8c10 test.c
```

## Related information

Compiler option **-C***cpu* (Select the CPU type)

Assembler option **--r8c** (Compile for R8C/tiny)

# --romstrings / --romconstants

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **C Compiler** entry and select **Code Generation**.

3. Enable the options **Keep strings in ROM** or **Keep constants in ROM**.

### Command line syntax

**--romstrings**

**--romconstants**

### Description

By default, strings literals and constants are copied from ROM to RAM at program startup. With these option you tell the compiler to keep string literals and / or constants in ROM. If you use these options, you can access these string literals or constants only with the `__rom` keyword.

### Example

To keep constant strings in ROM:

```
cm16c --romstrings test.c
```

### Related information

See also section 3.10, *Strings* in Chapter *C Language* of the *User's Guide*.

# –s (––source)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **C Compiler** entry and select **Miscellaneous**.

3. Enable the option **Merge C source code with assembly in output file (.src)**.

### Command line syntax

**–s**

**––source**

### 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.

### Example

```
cm16c –s test.c
```

The output file `test.src` contains the original C source lines as comments, besides the generated assembly code.

### Related information

–

# --static

### EDE

–

### Command line syntax

**--static**

### Description

With this option, the compiler treats external definitions at file scope (except for `main`) as if they were declared `static`. As a result, unused functions will be eliminated, and the alias checking algorithm assumes that objects with static storage cannot be referenced from functions outside the current module.

This option only makes sense when you specify all modules of an application on the command line.

### Example

```
cm16c --static module1.c module2.c module3.c
```

### Related information

–

# –t (––tradeoff)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **C Compiler** entry and select **Optimization**.

3. Select a trade–off level in the **Size/speed trade–off** box.

### Command line syntax

**–t**{**0**|**4**}

**––tradeoff=**{**0**|**4**}

### Default

**–t4**

### Description

If the compiler uses certain optimizations (option **–O**), you can use this option to specify whether the used optizations should opimize for more speed (regardless of code size) or for smaller code size (regardless of speed).

Default the compiler optimizes the selected optimizations for smaller code size (**–t4**).

If you have not used the option **–O**, the compiler uses default medium optimization, so you can still specify the option **–t**.

### Example

To set the trade–off level for the used optimizations:

```
cm16c –t0 test.c
cm16c ––tradeoff=0 test.c
```

The compiler uses the default medium optimization level and optimizes for code size rather than for speed.

### Related information

Compiler option **–O** (Specify optimization level)

# –U (––undefine)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **C Compiler** entry and select **Miscellaneous**.

3. Add the option **–U** to the **Additional compiler options** field.

### Command line syntax

**–U***macro_name*

**––undefine=***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.

However, 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__`:

```
cm16c -U__TASKING__ test.c
cm16c --undefine=__TASKING__ test.c
```

### Related information

Compiler option **–D** (Define preprocessor macro)

Section 3.8, *Predefined Macros*, in Chapter *Using the Compiler* of the *Users Guide*.

# –u (––uchar)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **C Compiler** entry and select **Miscellaneous**.

3. Enable the option **Treat 'char' variables as unsigned instead of signed**.

### Command line syntax

**–u**

**––uchar**

### Description

Treat 'character' type variables as 'unsigned character' variables. By default `char` is the same as specifying `signed char`. With **–u** `char` is the same as `unsigned char`.

### Example

With the following command `char` is treated as `unsigned char`:

```
cm16c –u test.c
cm16c ––uchar test.c
```

### Related information

–

# –V (––version)

## EDE

–

## Command line syntax

**–V**

**––version**

## Description

Display version information. The compiler ignores all other options or input files.

## Example

```
cm16c –v
cm16c ––version
```

The compiler does not compile any files but displays the following version information:

```
TASKING M16C C compiler          vxx.yrz Build nnn
Copyright 2002–year Altium BV    Serial# 00000000
```

## Related information

–

# –w (––no-warnings)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **C Compiler** entry and select **Diagnostics**.

3. Enable one of the options **Report all warnings**, **Suppress all warnings**, or **Suppress specific warnings**.

   *If you select **Suppress specific warnings**:*

4. Enter the numbers, separated by commas, of the warnings you want to suppress.

### Command line syntax

**–w**[*nr*]

**––no–warnings**[**=***nr*]

### Description

With this option you can suppresses all warning messages or specific warning messages.

- If you do not specify this option, all warnings are reported.
- If you specify this option but without numbers, all warnings are suppressed.
- If you specify this option with a number, only the specified warning is suppressed. You can specify the option **–w** multiple times.

### Example

To suppress all warnings:

```
cm16c test.c –w
cm16c test.c ––no–warnings
```

To suppress warnings 135 and 136:

```
cm16c test.c –w135 –w136
cm16c test.c ––no–warnings=135 ––no–warnings=136
```

### Related information

Compiler option **--warnings-as-errors** (Treat warnings as errors)

# --warnings-as-errors

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **C Compiler** entry and select **Diagnostics**.
3. Enable the option **Treat warnings as errors**.

### Command line syntax

**--warnings-as-errors**

### Description

With this option you tell the compiler to treat warnings as errors.

### Example

```
cm16c --warnings-as-errors test.c
```

When a warning occurs, the compiler considers it as an error.

### Related information

Compiler option **-w** (suppress some or all warnings)

## 4.2 ASSEMBLER OPTIONS

This section lists all assembler options.

### *Options in EDE versus options on the command line*

Most command line options have an equivalent option in EDE but some options are only available on the command line. If there is no equivalent option in EDE, you can specify a command line option in EDE as follows:

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Assembler** entry and select **Miscellaneous**.

3. Enter one or more command line options in the **Additional assembler options** field.

Be aware that some command line options are not useful in EDE or just do not have any effect. For example, the option **–V** displays version header information and has no effect in EDE.

### *Short and long option names*

Options 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:

```
asm16c -Lmx test.src
asm16c --list-format=+macro,+macro-expansion test.src
```

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

# –? (––help)

## EDE

–

## Command line syntax

**–?**

**––help**[**=options**]

## Description

Displays an overview of all command line options. When you specify the **options** argument, a list with option descriptions is displayed.

## Example

The following invocations all display a list of the available command line options:

```
asm16c –?
asm16c ––help
asm16c
```

The following invocation displays extended information about all options:

```
asm16c ––help=options
```

## Related information

–

# –C (––cpu)

## EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Processor** entry and select **Processor Definition**.

3. Select a processor from the **Select processor** box.

## Command line syntax

**–C***cpu*

**––cpu=***cpu*

## Description

With this option you define the target processor for which you create your application.

Based on the target processor, the assembler includes a *special function register file* `reg`*`cpu`*`.sfr`. This is an include file written in C syntax which is shared by the compiler, assembler and debugger. Once the assembler reads an SFR file you can reference the special function registers (SFR) and bits within an SFR using symbols defined in the SFR file.

When you select an R8C target, you always must use the assembler option **––r8c** as well !

## Example

To assemble the file `test.c` for the m30624 processor and use the SFR file `regm30624a.sfr`:

```
asm16c –Cm30624a test.src
asm16c ––cpu=m30624a test.src
```

To avoid conflicts, make sure you specify the same target processor as you did for the compiler.

### Related information

Compiler option **–C** (Select the CPU type)
Control program option **–C** (Select target CPU)

Assembler option **––r8c** (Assemble for R8C/tiny)

# –c (––case-insensitive)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Assembler** entry and select **Miscellaneous**.

3. Disable the option **Case sensitive assembly**.

### Command line syntax

**–c**

**––case–insensitive**

### 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 in labels and user–defined symbols as different characters. Note that instruction mnemonics, register names, directives and controls are always treated case insensitive.

*Disabling* the option **Assemble case sensitive** in EDE is the same as specifying the option **–c** on the command line.

Assembly source files that are generated by the compiler must always be assembled case sensitive. When you are writing your own assembly code, you may want to specify the case insensitive mode.

### Example

To assemble case insensitive:

```
asm16c –c test.src
asm16c ––case–insensitive test.src
```

The assembler considers upper and lower case characters as being the same. So, for example, the label `LabelName` is the same label as `labelname`.

### Related information

Linker option **––case–sensitive** (Link case insensitive)

# --check

### EDE

1. In the project window, select the file you want to check.
2. From the **Build** menu, select **Check Syntax**.

### 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.

### Example

To check for syntax errors, without generating code:

```
asm16c --check test.src
```

### Related information

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

# –D (––define)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Assembler** entry and select **Preprocessing**.

3. Click on an empty **Macro** field and enter a macro name.

4. Optionally, click in the **Definition** field and enter a definition.

   *Use commas to separate multiple macro definitions.*

### Command line syntax

**–D***macro_name*[**=***macro_definition*]

**––define=***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. On the command line you can use the option **–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 **–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 `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 C program with conditional code to compile 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:

```
asm16c -DDEMO test.src
asm16c -DDEMO=1 test.src

asm16c --define=DEMO test.src
asm16c --define=DEMO=1 test.src
```

Note that all four invocations have the same effect.

### Related information

Assembler option **–f** (Specify an option file)

Section 4.10.5, *Conditional Assembly*, in Chapter *Assembly Language* of the *User's Guide*.

# –-diag

### EDE

1. In the **Help** menu, enable the option **Show Help on Tool Errors**.
2. In the **Build** tab of the **Output** window, double–click on an error or warning message.

   *A description of the selected message appears.*

### Command line syntax

**–-diag=**[*format***:**]{**all** | *number*[**,***number*]... }

Optionally, you can use one of the following display formats (*format*):

| | |
|---|---|
| **text** | The default is plain text |
| **html** | Display explanation in HTML format |
| **rtf** | Display explanation in RTF format |

### 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.

To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of *all* error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas.

With this option the assembler does not assemble any files.

### Example

To display an explanation of message number 241, enter:

```
asm16c --diag=241
```

This results in the following message and explanation:

```
W241: additional input files will be ignored

The assembler supports only a single input file. All
other input files are ignored.
```

To write an explanation of all errors and warnings in HTML format to file `aserrors.html`, enter:

```
asm16c --diag=html:all > aserrors.html
```

### Related information

–

# –-emit-locals

### EDE

–

### Command line syntax

**–-emit-locals**

### Description

With this option the assembler also emits local symbols to the object file's symbol table. Normally, only global symbols are emitted.

### Example

To emit local symbols, enter:

```
asm16c --emit-locals test.src
```

### Related information

–

# –-error-file

### EDE

–

### 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:

```
asm16c --error-file=errors.ers test.src
```

### Related information

Assembler option **–-warnings–as–errors** (Treat warnings as errors)

# –f (––option-file)

## EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Assembler** entry and select **Miscellaneous**.

3. Add the option **–f** to the **Additional assembler options** field.

In EDE you can save your options in a file and restore them to call the assembler with those options:

- From the **Project** menu, select **Save Options...** or **Load Options...**

Be aware that when you specify the option **–f** in the **Additional assembler options** field, the options are *added* to the assembler options you have set in the Project Options dialog. Only in extraordinary cases you may want to use them in combination.

## Command line syntax

**–f** *file*,...

**––option–file=***file*,...

## Description

Instead of typing all options on the command line, you can create an option file which contains all options and files you want to specify. With this option you specify the option file to the assembler.

Use an option file when the length of the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option **–f** multiple times.

### *Format of an option file*

- Multiple command line 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:

```
-Cm30624
test.src
```

Specify the option file to the assembler:

```
asm16c -f myoptions
asm16c --option-file=myoptions
```

This is equivalent to the following command line:

```
asm16c -Cm30624 test.src
```

### Related information


–

# –g (––debug-info)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Assembler** entry and select **Debug**.

3. Enable one or more debug options.

You cannot use **Assembly source line information** and **Pass HLL debug information** simultaneously.

### Command line syntax

**–g**[*flag*]

**––debug–info**[**=***flag*]

You can set the following flags:

| | | |
|---|---|---|
| **a**/**A** | (**+**/**–asm**) | Assembly source line information |
| **h**/**H** | (**+**/**–hll**) | Pass HLL debug information |
| **l**/**L** | (**+**/**–local**) | Local symbols debug information |
| **s**/**S** | (**+**/**–smart**) | Smart debug information |

### Default

**–gs**

### Description

With this option you tell the assembler to generate debug information. If you do not use this option or if you specify **–g** without any flags, the default is **–gs**.

You cannot specify **–gah**. Either the assembler generates assembly source line information, or it passes HLL debug information.

When you specify **–gs**, 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 **–gAhL**). If not, the assembler generates assembly source line information and local symbols debug information (same as **–gaHl**).

With **–gAHLS** the assembler does not generate any debug information.

### Example

To disable symbolic debug information, turn all flags off:

```
asm16c -gAHLS test.src
asm16c --debug-info=-asm,-hll,-local,-smart test.src
```

To enable smart debugging, enter:

```
asm16c -gs test.src
asm16c --debug-info=+smart test.src
```

### Related information


–

# –H (––include-file)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Assembler** entry and select **Preprocessing**.

3. Enter the name of the file in the **Include this file before source** field.

### Command line syntax

**–H***file*,...

**––include–file=***file*,...

### Description

With this option you include one extra file at the beginning of the assembly source file, before other includes. This is the same as specifying `INCLUDE 'file'` at the beginning of your assembly sources.

### Example

```
asm16c –Hmyinc.inc test1.src
asm16c ––include–file=myinc test1.src
```

The file `myinc.inc` is included at the beginning of `test1.src` before it is assembled.

### Related information

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

Section 6.5, *How the Assembler Searches Include Files*, in Chapter *Using the Assembler* of the *User's Guide*.

# –I (––include-directory)

### EDE

1. From the **Project** menu, select **Directories...**

   *The Directories dialog appears.*

2. Enter one or more search paths in the **Include Files Path** field.

### Command line syntax

**–I***path*,...

**––include–directory=***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 absolute pathname, if specified in the `INCLUDE` directive. Or, if no path or a relative path is specified, the same directory as the source file.
2. The directories that are specified with this option.
3. The directories that are specified in the environment variable ASM16CINC when the product was installed.
4. The default `include` directory relative to the installation directory.

### Example

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

```
INCLUDE 'myinc.inc'
```

You can call the assembler as follows:

```
asm16c –Ic:\proj\include test.src
asm16c ––include–directory=c:\proj\include test.src
```

First the assembler looks in the directory where `test.src` is located for the file `myinc.inc`. If it does not find the file, it looks in the directory `c:\proj\include` for the file `myinc.inc` (this option).

### Related information

Section 6.5, *How the Assembler Searches Include Files*, in Chapter *Using the Assembler* of the *User's Guide*.

Section 1.3.2, *Configuring the Command Line Environment*, in Chapter *Software Installation* of the *User's Guide*.

Assembler option **–H** (Include file at the start of the input files)

Compiler option **–I** (Add directory to include file search path)

# –i (––symbol-scope)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Assembler** entry and select **Miscellaneous**.

3. Add the option **`-i`** to the **Additional assembler options** field.

### Command line syntax

**–i**{**g**|**l**}

**––symbol–scope=**{**global**|**local**}

### Default

**–il**

### Description

With this option you tell the assembler how to treat symbols that you have not specified explicitly as global or local.

By default the assembler treats all symbols as local symbols unless you have defined them explicitly as global.

### Example

```
asm16c –ig test.src
asm16c ––symbol–scope=global test.src
```

The assembler treats all symbols as global symbols unless they are defined as local symbols in the assembly source file.

### Related information


–

# –k (––keep-output-files)

## EDE

EDE always removes the assembler output file when errors occur during assembling.

## Command line syntax

**–k**

**––keep–output–files**

## Description

If an error occurs during assembly, the resulting `.obj` file may be incomplete or incorrect. With this option you keep the generated object file (`.obj`) when an error occurs.

By default the assembler removes the generated object file (`.obj`) 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.

## Example

```
asm16c –k test.src
```

When an error occurs during assembly, the generated output file `test.obj` will *not* be removed.

## Related information

Assembler option **––warnings–as–errors** (Treat warnings as errors)

# –L (––list-format)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Assembler** entry and select **List File**.

3. Select **Custom list file generation** from the **List file generation** box.

4. Enable the options to include that information in the list file.

### Command line syntax

**–L***flags*

**––list–format=***flags*

You can set the following flags:

| | | |
|---|---|---|
| **0** | | same as **–LCDEGIMNPQRSVWXY** |
| **1** | | same as **–Lcdegimnpqrsvwxy** |
| **c**/**C** | (**+**/**–control**) | Assembler control lines |
| **d**/**D** | (**+**/**–section**) | Section directives |
| **e**/**E** | (**+**/**–symbol**) | Symbol definition directives |
| **g**/**G** | (**+**/**–generic–expansion**) | Generic instruction expansion |
| **i**/**I** | (**+**/**–generic**) | Generic instructions |
| **m**/**M** | (**+**/**–macro**) | Macro definitions |
| **n**/**N** | (**+**/**–empty–line**) | Empty source lines |
| **p**/**P** | (**+**/**–conditional**) | Conditional assembly |
| **q**/**Q** | (**+**/**–equate**) | Assembler EQU and SET directives |
| **r**/**R** | (**+**/**–relocations**) | Relocation characters ('r') |
| **s**/**S** | (**+**/**–hll**) | HLL symbolic debug information |
| **v**/**V** | (**+**/**–equate–values**) | Assembler EQU and SET values |
| **w**/**W** | (**+**/**–wrap–lines**) | Wrapped source lines |
| **x**/**X** | (**+**/**–macro–expansion**) | Macro expansions |
| **y**/**Y** | (**+**/**–cycle–count**) | Cycle counts |

### Default

**–LcDEGiMnPqrsVWXy**

### Description

With this option you specify which information you want to include in the list file. Use this option in combination with the option **–l** (**––list–file**).

If you do not specify this option, the assembler uses the default: **–LDEGiMnPqrsVWXy**.

With option **–tl**, the assembler also writes section information to the list file.

### Example

```
asm16c –l –Ldm test.src
asm16c ––list–file ––list–format=+section,+macro
     test.src
```

The assembler generates a list file that includes all default information plus section directives and macro definitions.

### Related information

Assembler option **–l** (Generate list file)
Assembler option **–tl** (Display section information in list file)

Linker option **–M** (Generate map file)

Section 5.1, *Assembler List File Format*, in Chapter *List File Formats*.

# –l (––list-file)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Assembler** entry and select **List File**.

3. Enable the option **List file generation**.

### Command line syntax

**–l**

**––list–file**

### Description

With this option you tell the assembler to generate a list file. A list file shows the generated object code and the relative addresses. Note that the assembler generates a relocatable object file with relative addresses.

### Example

To generate a list file with the name `test.lst`, enter:

```
asm16c -l test.src
asm16c --list-file test.src
```

### Related information

Assembler option **–L** (List file formatting options)

Linker option **–M** (Generate map file)

Section 5.1, *Assembler List File Format*, in Chapter *List File Formats*.

# –m (––preprocessor-type)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Assembler** entry and select **Miscellaneous**.

3. Add the option **–m** to the **Additional assembler options** field.

### Command line syntax

**–m**{**n** | **t**}

**––preprocessor–type=**{**none** | **tasking**}

### Default

**–mt**

### Description

With this option you select the preprocessor that the assembler will use. Default, the assembler uses the TASKING preprocessor.

When the assembly source file does not contain any preprocessor symbols, you can specify the assembler not to use a preprocessor.

### Example

```
asm16c test.src
asm16c –mt test.src
asm16c ––preprocessor=tasking test.src
```

These invocations have the same effect: the assembler preprocesses the file `test.src` with the TASKING preprocessor.

### Related information


–

# –O (––optimize)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Assembler** entry and select **Miscellaneous**.

3. Enable or disable the optimization suboptions.

### Command line syntax

**–O***flags*

**––optimize=***flags*

You can set the following flags:

| | | |
|---|---|---|
| **a**/**A** | (**+**/**–align**) | Speed optimization by means of instruction alignment |
| **g**/**G** | (**+**/**–generics**) | Allow generic instructions |
| **j**/**J** | (**+**/**–jumpchains**) | Jumpchains |
| **s**/**S** | (**+**/**–instr–size**) | Optimize instruction size |

### Default

**–OAgs**

### Description

With this option you can control the level of optimization. If you do not use this option, **–OAgJs** is the default.

### Example

The following invocations are equivalent and result all in the default optimizations:

```
asm16c test.src
asm16c –OAgJs test.src
asm16c ––optimize=–align,+generics,–jumpchains,
     +instr–size test.src
```

### Related information

Section 6.3, *Assembler Optimizations*, in Chapter *Using the Assembler* of the *User's Guide*.

# –o (––output)

## EDE

–

## Command line syntax

**–o***file*

**––output=***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`.

EDE names the output file always after the assembly source file.

## Example

```
asm16c –o relobj.obj asm.src
asm16c ––output=relobj.obj asm.src
```

The assembler creates the file `relobj.obj` for the assembled file `asm.src`.

Without the option **–o**, like EDE, the assembler uses the name of the input file and creates `asm.obj`.

## Related information

–

# --r8c

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Processor** entry and select **Processor Definition**.

3. Select a processor from the **Select processor** box.

### Command line syntax

**--r8c**

### Description

By default, the assembler generates code for the M16C/60 core. With this option you tell the assembler to assemble for R8C/tiny core. You must use this option always (and only then) when you select an R8C target with the assembler option **-C***cpu*. In EDE this option is automatically enabled when you select an R8C target.

When you compile for the R8C/tiny, code sections are always placed in near memory.

To avoid conflicts, make sure you only specify this option if you specified it also to the compiler.

### Example

```
asm16c --r8c --Cr8c10 test.src
```

### Related information

Assembler option **-C***cpu* (Select the CPU type)

Compiler option **--r8c** (Compile for R8C/tiny)

# –t (––section-info)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Assembler** entry and select **Miscellaneous**.

3. Enable the option **Generate section summary**.

   *EDE always writes the section information to the list file.*

### Command line syntax

**–t***flags*

**––section–info=***flags*

You can set the following flags:

| | | |
|---|---|---|
| **c**/**C** | (**+**/**–console**) | Display section information on `stdout`. |
| **l**/**L** | (**+**/**–list**) | Write section information to the list file. |

### Description

With this option you tell the assembler to display section information. For each section its memory space, size, total cycle counts and name is listed on `stdout` and/or in the list file.

The cycle count consists of two parts: the total accumulated count for the section and the total accumulated count for all repeated instructions. In the case of nested loops it is possible that the total supersedes the section total.

With **–tl**, the assembler writes the section information to the list file. You must specify this option in combination with the option **–l** (generate list file).

### Example

```
asm16c -l -tcl test.src
asm16c --list-file --section-info=+console,+list
      test.src
```

The assembler generates a list file and writes the section information to this file. The section information is also displayed on `stdout`.

```
Section summary:

  REL                   2       6  test_CO
  REL                  20          test_INI_BI
  REL                   c          test_INI_DA
  ABS (00000010)        2          test_CLR_DA_00000010
  REL                   2          test_RO_DA
```

### Related information

Assembler option **–l** (Generate list file)

# –V (––version)

### EDE

–

### Command line syntax

**–V**

**––version**

### Description

Display version information. The assembler ignores all other options or input files.

### Example

```
asm16c -V
asm16c --version
```

The assembler does not assemble any files but displays the following version information:

```
TASKING M16C Assembler                  vx.yrz Build nnn
Copyright 2003-years Altium BV          Serial# 00000000
```

### Related information

–

# –v (––verbose)

### EDE

–

### Command line syntax

**–v**

**––verbose**

### Description

With this option you put the assembler in verbose mode. The assembler prints the filenames and the assembly passes while it processes the files so you can monitor the current status of the assembler.

### Example

```
asm16c –v test.src
asm16c ––verbose test.src
```

### Related information

–

# –w (––no-warnings)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Assembler** entry and select **Diagnostics**.

3. Enable one of the options **Report all warnings**, **Suppress all warnings**, or **Suppress specific warnings**.

   *If you select **Suppress specific warnings**:*

4. Enter the numbers, separated by commas, of the warnings you want to suppress.

### Command line syntax

**–w**[*nr*,...]

**––no–warnings**[**=***nr*,...]

### Description

With this option you can suppresses all warning messages or specific warning messages.

- If you do not specify this option, all warnings are reported.
- If you specify this option but without numbers, all warnings are suppressed.
- If you specify this option with a number, only the specified warning is suppressed. You can specify the option **–w** multiple times.

### Example

To suppress all warnings:

```
asm16c –w test.src
asm16c ––no–warnings test.src
```

To suppress warnings 135 and 136:

```
asm16c –w135,136 test.src
asm16c ––no–warnings=135,136 test.src
```

### Related information

Assembler option **--warnings-as-errors** (Treat warnings as errors)

# –-warnings-as-errors

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Assembler** entry and select **Diagnostics**.

3. Enable the option **Treat warnings as errors**.

### Command line syntax

**–-warnings–as–errors**

### Description

With this option you tell the assembler to treat warnings as errors.

### Example

```
asm16c –-warnings–as–errors test.src
```

When a warning occurs, the assembler considers it as an error. No object file is generated, unless you specify option **–k** (**–-keep–output–files**).

### Related information

Assembler option **–w** (suppress some or all warnings)

## 4.3 LINKER OPTIONS

EDE uses a *makefile* to build your entire project. This means that you cannot run the linker separately. However, you can set options specific for the linker.

### *Options in EDE versus options on the command line*

Most command line options have an equivalent option in EDE but some options are only available on the command line.

See section 4.4, *Control Program Options*.

If there is no equivalent option in EDE, you can specify a command line option in EDE as follows:

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Linker** entry and select **Miscellaneous**.

3. Enter one or more command line options in the **Additional linker options** field.

Be aware that some options are not useful in EDE or just will not have any effect. For example, the option **`-k`** keeps files after an error occurred. When you specify this option in EDE, it will have no effect because EDE *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:

```
lkm16c -mfkl test.obj
lkm16c --map-file-format=+files,+link,+locate test.obj
```

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

# –? (––help)

## EDE

–

## 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:

```
lkm16c –?
lkm16c ––help
lkm16c
```

## Related information

–

# –c (––chip-output)

## EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Linker** entry and select **Output Format**.

3. Enable one or more output formats.

## Command line syntax

**–c**[*basename*]:*format*[**:***addr_size*]**,**...

**––chip–output=**[*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** and **4** (default). For Motorola S you can specify: **2** (S1 records), **3** (S2 records, default) or **4** bytes (S3 records). In EDE you cannot specify the address size because EDE always uses the default values.

## Description

With this option you specify the Intel Hex or Motorola S–record output format for loading into a PROM–programmer. The linker generates a file for each ROM memory defined in the LSL file, where sections are located:

```
memory memname
{  type=rom;  }
```

The name of the file is the name of the EDE project or, on the command line, the name of the memory space that was emitted with extension `.hex` or `.s`. Optionally you can specify a *basename* which prepends the generated file name.

### Examples

To generate Intel Hex output files for each defined memory, enter the following on the command line:

```
lkm16c -cmyfile:IHEX test1.obj
lkm16c --chip-output=myfile:IHEX test1.obj
```

This generates the file `myfile_`*`memspace`*`.hex`.

### Related information

Linker option **-o** (output file)

Section 6.2, *Motorola S–Record Format*,
Section 6.3, *Intel Hex Record Format*, in Chapter *Object File Formats*.

# –-case-insensitive

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Linker** entry and select **Libraries**.

3. Disable the option **Link case sensitive**.

### Command line syntax

**–-case–insensitive**

### Description

With this option you tell the linker not to distinguish between upper and lower case characters. By default the linker considers upper and lower case characters as different characters.

*Disabling* the option **Link case sensitive** in EDE is the same as specifying the option **–-case–insensitive** on the command line.

Assembly source files that are generated by the compiler must always be assembled and thus linked case sensitive. When you have written your own assembly code and specified to assemble it case insensitive, you must also link the `.obj` file case insensitive.

### Example

To link case insensitive:

```
lkm16c –-case–insensitive test.obj
```

The linker considers upper and lower case characters as being the same. So, for example, the label `LabelName` is considered the same label as `labelname`.

### Related information

Assembler option **–c** (Assemble case insensitive)

# –D (––define)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog box appears.*

2. Expand the **Linker** entry and select **Miscellaneous**.

3. Add the option **–D** to the **Additional linker options** field.

### Command line syntax

**–D***macro_name*[**=***macro_definition*]

**––define=***macro_name*[**=***macro_definition*]

### Description

With this option you can define a macro and specify it to the linker 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: you can use the option **–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 **–f***file*.

Define *macro* to the preprocessor, as in #define. Any number of symbols can be defined. The definition can be tested by the preprocessor with `#if`, `#ifdef` and `#ifndef`, for conditional locating.

### Example

To define the RESET vector, interrupt table start address and trap table start address which is used in the linker script file `m16c.lsl`, enter:

```
lkm16c test.obj –otest.elf –dm16c.lsl
  –DRESET=0xa0000000 –DINTTAB=0xa00f0000
  –DTRAPTAB=0xa00f2000
```

or:

```
lkm16c test.obj –otest.elf ––lsl–file=m16c.lsl
  ––define=RESET=0xa0000000 ––define=INTTAB=0xa00f0000
  ––define=TRAPTAB=0xa00f2000
```

### Related information

Linker option **–f** (Name of invocation file)

# –d (––lsl-file)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Linker** entry and select **Miscellaneous**.

3. Select **Use project specific linker script file** and specify a name.

### Command line syntax

**–d***file*
**––lsl–file=***file*

### Description

With this option you specify a linker script file to the linker. If you do not specify this option, the linker uses a default script file. You can specify the existing file `target.lsl` or the name of a manually created 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.

The 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 in the system.
- the section layout definition describes how to locate sections in memory.

### Example

To read linker script file information from file `mylslfile.lsl`:

```
lkm16c –dmylslfile.lsl test.obj
lkm16c ––lsl–file=mylslfile.lsl test.obj
```

### Related information

Linker option **––lsl–check** (Check LSL file(s) and exit)

Section 7.6, *Controlling the Linker with a Script* in Chapter *Using the Linker* of the *User's Guide*.

# –-diag

### EDE

1. In the **Help** menu, enable the option **Show Help on Tool Errors**.
2. In the **Build** tab of the **Output** window, double–click on an error or warning message.

   *A description of the selected message appears.*

### Command line syntax

**–-diag=**[*format*:]{**all** | *number*[,*number*]... }

Optionally, you can use one of the following display formats (*format*):

| | |
|---|---|
| **text** | The default is plain text |
| **html** | Display explanation in HTML format |
| **rtf** | Display explanation in RTF format |

### 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.

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 any files.

### Example

To display an explanation of message number 106, enter:

```
lkm16c –-diag=106
```

This results in the following message and explanation:

```
E106: unresolved external: message

The linker could not resolve all external symbols. This is an
error when the incremental linking option is disabled. The
<message> indicates the symbol that is unresolved.
```

To write an explanation of all errors and warnings in HTML format to file `lerrors.html`, enter:

```
lkm16c --diag=html:all > lerrors.html
```

### Related information

Section 7.9, *Linker Error Messages* in Chapter *Using the Linker* of the *User's Guide*.

# –e (––extern)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Linker** entry and select **Miscellaneous**.

3. Add the option **–e** in the **Additional linker options** field.

### Command line syntax

**–e** *symbol*

**––extern=***symbol*

### Description

With this option you force the linker to consider the given symbol as an undefined reference. The linker tries to resolve this symbol by extracting the corresponding symbol definition from a library. If the symbol is defined in an object file, this option has no influence on the link process.

Suppose you are linking from a library. Because the library itself already has been compiled and assembled, the linker does not find any unresolved symbols. Hence, the linker will not extract any module from the library. When you force a symbol to be undefined, the linker extracts those modules that contain the symbol.

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:

```
lkm16c mylib.a
```

Nothing is linked and no output file will be produced, because there are no unresolved symbols when the linker searches through `mylib.a`.

```
lkm16c -e __START mylib.a
lkm16c --extern=__START mylib.a
```

In this case the linker searches for the symbol `__START` in the library and (if found) extracts the object that contains `__START`, the startup code. If this module contains new unresolved symbols, the linker looks again in `mylib.a`. This process repeats until no new unresolved symbols are found.

### Related information

Section 7.4.1, *Specifying Libraries to the Linker*, in Chapter *Using the Linker* of the *User's Guide*.

# –-error-file

### EDE

–

### 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 will be named after the input file with extension `.elk`.

### Example

```
lkm16c --error-file=my.elk test.obj
```

The linker writes error messages to the file `my.elk` instead of `stderr`.

### Related information

Linker option **--warnings-as-errors** (Treat warnings as errors)

# –f (––option-file)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Linker** entry and select **Miscellaneous**.

3. Add the option **–f** to the **Additional linker options** field.

In EDE you can save your options in a file and restore them to call the linker with those options:

1. From the **Project** menu, select **Save Options...** or **Load Options...**

Be aware that when you specify the option **–f** in the **Additional linker options** field, the options are *added* to the linker options you have set in the Project Options dialog. Only in extraordinary cases you may want to use them in combination.

### Command line syntax

**–f** *file*,...

**––option–file=***file*,...

### Description

Instead of typing all options on the command line, you can create an option file which contains all options and files you want to specify. With this option you specify the option file to the linker.

Use an option file when the length of the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option **–f** multiple times.

#### *Format of an option file*

- Multiple command line 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:

| | |
|---|---|
| `-Mmymap` | (generate a map file) |
| `test.obj` | (input file) |
| `-Lc:\mylibs` | (additional search path for system libraries) |

Specify the option file to the linker:

```
lkm16c -f myoptions
lkm16c --option-file=myoptions
```

This is equivalent to the following command line:

```
lkm16c -Mmymap test.obj -Lc:\mylibs
```

### Related information


–

# –-first-library first

### EDE

–

### 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 at 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.

With this option, you tell the linker to scan the libraries from left to right, and extract the symbol from the first library where the linker finds it.

### Example:

```
lkm16c –-first–library–first a.a test.obj b.a
```

If the file `test.obj` calls a function which is both present in `a.a` and `b.a`, normally the function in `b.a` would be extracted. With this option the linker first tries to extract the symbol from the first library `a.a`.

### Related information

Linker option **–-no–rescan** (Do not rescan libraries)

# –I (––include-directory)

## EDE

–

## Command line syntax

**–I***path*,...

**––include–directory=***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 files are located (only for #include files that are enclosed in "")
2. The path that is specified with this option.
3. The default `..\include.lsl` directory relative to the installation directory.

## Example

Suppose that the LSL file `lslfile.lsl` contains the following lines:

```
#include "mypart.lsl"
```

You can call the linker as follows:

```
lkm16c –Imyinclude –dlslfile.lsl test.obj
lkm16c ––include–directory=myinclude
       ––lsl–file=lslfile.lsl test.obj
```

First the linker looks in the directory where `lslfile.lsl` is located for the file `mypart.lsl`. If it does not find the file, it looks in `myinclude` subdirectory relative to the current directory for the file `mypart.lsl` (this option). Finally it looks in the directory `..\include.lsl`.

## Related information

Linker option **–d** (Linker script file)

# –i (––user-provided-initialization-code)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Linker** entry and select **Miscellaneous**.

3. Disable the option **Use standard copy–table for initialization**.

### Command line syntax

**–i**

**––user–provided–initialization–code**

### 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–item–compression' and 'copytable–compression' optimizations are automatically disabled when you enable this option.

### Example:

To link with your own startup code:

```
lkm16c –i test.obj

lkm16c ––user–provided–initialization–code test.obj
```

### Related information

–

# –k (––keep-output-files)

### EDE

EDE always removes the output files when errors occur during linking.

### Command line syntax

**–k**

**––keep–output–files**

### 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 files when an error occurs. This is useful when you use the make utility **mkm16c**. 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 the error(s) do not result in a corrupt output file, or when you want to inspect the output file, or send it to Altium support.

### Example

```
lkm16c –k test.obj
lkm16c ––keep–output–files test.obj
```

When an error occurs during linking, the generated output file `test.elf` will *not* be removed.

### Related information

–

# –L (––library-directory / ––ignore-default-library-path)

### EDE

1. From the **Project** menu, select **Directories...**

   *The Directories dialog appears.*

2. Add a pathname in the **Library Files Path** field.

3. In the **Library Files Path** field, add the pathnames of the directories where the linker should look for library files.

   *If you enter multiple paths, separate them with a semicolon (;).*

### Command line syntax

**–L***path*,...
**––library–directory=***path*,...

**–L**
**––ignore–default–library–path**

### Description

With this option you can specify the path(s) where your system libraries, specified with the **–l** option, are located. If you want to specify multiple paths, use the option **–L** for each separate path.

The default path is `$(PRODDIR)\lib\m16c`.

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 LIBM16C. 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 **–l** option is:

1. The path that is specified with the **–L** option.

2. The path that is specified in the environment variable LIBM16C when the product was installed.

3. The default directory `$(PRODDIR)\lib\m16c`.

### Example

Suppose you call the linker as follows:

```
lkm16c test.obj -Lc:\mylibs -lcs
```

First the linker looks in the directory `c:\mylibs` for library `libcs.a` (this option).

If it does not find the requested libraries, it looks in the directory that is set with the environment variable LIBM16C.

Then the linker looks in the default directory `$(PRODDIR)\lib\m16c` for libraries.

### Related information

Linker option **–l** (Link system library)

Section 7.4.2, *How the Linker Searches Libraries*, in Chapter *Using the Linker* of the *User's Guide*.

Section 1.3.2, *Configuring the Command Line Environment*, in Chapter *Software Installation* of the *User's Guide*.

# –l (––library)

## EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Linker** entry and select **Libraries**.

3. Enable the option **Link default C libraries**.

## Command line syntax

**–l***name*

**––library=***name*

## Description

With this option you tell the linker to search also in system library `lib`*`name`*`.a`, where *name* is a string. The linker first searches for system libraries in any directories specified with **–L***path*, then in the directories specified with the environment variable LIBM16C, unless you used the option **–L** without a directory.

## Example

To search in the system library `libfps.a` (floating–point library):

```
lkm16c test.obj mylib.a –lfps
lkm16c test.obj mylib.a ––library=fps
```

The linker links the file `test.obj` and first looks in `mylib.a` (in the current directory only), then in the system library `libfps.a` to resolve unresolved symbols.

## Related information

Linker option **–L** (Additional search path for system libraries)

Section 7.4.1, *Specifying Libraries to the Linker*, in Chapter *Using the Linker* of the *User's Guide*.

# --link-only

### EDE

–

### Command line syntax

**--link-only**

### Description

With this option you suppress the locating phase. The linker stops after linking. The linker complains if any unresolved references are left.

### Example:

```
lkm16c --link-only hello.obj
```

The linker checks for unresolved symbols and creates the file `hello.eln`.

### Related information

Control program option **-cl** (Stop after linking)

# --lsl-check

### EDE

–

### 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.

### Example:

To check the LSL file(s) and exit:

```
lkm16c --lsl-check --lsl-file=mylslfile.lsl
```

### Related information

Linker option **-d** (Linker script file)
Linker option **--lsl-dump** (Dump LSL info)

Chapter 7, *Linker Script Language*.

# --lsl-dump

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Linker** entry and select **Miscellaneous**.

3. Enable the option **Dump processor and memory info from LSL file**.

### Command line syntax

**--lsl-dump**[=*file*]

### Description

With this option you tell the linker to dump the LSL part of the map file in a separate file, independent of the **-M** (generate map file) option. If you do not specify a filename, the file `lkm16c.ldf` is used.

### Example

```
lkm16c --lsl-dump=mydump.ldf test.obj
```

The linker dumps the processor and memory info from the LSL file in the file `mydump.ldf`.

### Related information

Linker option **-m** (Map file formatting options)

# –M (––map-file)

## EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Linker** entry and select **Map File**.

3. Enable the option **Generate a linker map file (.map)**.

## Command line syntax

**–M**[*file*]

**––map–file**[**=***file*]

## Description

With this option you tell the linker to generate a linker map file. If you do not specify a filename, the linker uses the same basename as the output file with the extension `.map`.

A linker map file is a text file that shows how the linker has mapped the sections and symbols from the various object files (`.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 the option **–m** (map file formatting) you can specify which parts you want to place in the map file.

## Example

To generate a map file (`test.map`):

```
lkm16c –Mtest.map test.obj
lkm16c ––map–file=test.map test.obj
```

The control program by default tells the linker to generate a map file.

## Related information

Linker option **–m** (Map file formatting options)

Section 5.2, *Linker Map File Format*, in Chapter *List File Formats*.

# –m (––map-file-format)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Linker** entry and select **Map File**.

3. Enable the options to include several kinds of information in the map file.

### Command line syntax

**–m***flags*

**––map–file–format=***flags*

You can set the following flags:

| | | |
|---|---|---|
| **0** | same as **–mcfkLMoQrSU** | (link info) |
| **1** | same as **–mCfKlmoQRSU** | (locate info) |
| **2** | same as **–mcfklmoqrsu** | (all info) |

| | | |
|---|---|---|
| **c**/**C** | (**+**/**–callgraph**) | Call graph info |
| **f**/**F** | (**+**/**–files**) | Processed files info |
| **k**/**K** | (**+**/**–link**) | Link result info |
| **l**/**L** | (**+**/**–locate**) | Locate result info |
| **m**/**M** | (**+**/**–memory**) | Memory usage info |
| **o**/**O** | (**+**/**–overlay**) | Overlay info |
| **q**/**Q** | (**+**/**–statics**) | Module local symbols |
| **r**/**R** | (**+**/**–crossref**) | Cross references info |
| **s**/**S** | (**+**/**–lsl**) | Processor and memory info |
| **u**/**U** | (**+**/**–rules**) | Locate rules |

### Default

**–mCfklMORSU**

### Description

With this option you specify which information you want to include in the map file. Use this option in combination with the option **–M** (**––map–file**). If you do not specify this option, the linker uses the default: **–mCfklMORSU**

### Example

```
lkm16c -Mtest.map -mFr test.obj
lkm16c --map-file=test.map
       --map-file-format=+crossref,-files test.obj
```

The linker generates the map file `test.map` that includes all default information plus the cross reference part, but not the processed files part.

### Related information

Linker option **-M** (Generate map file)

Section 5.2, *Linker Map File Format*, in Chapter *List File Formats*.

# –-misra-c-report

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **C Compiler** entry and select **MISRA C**.
3. Select a MISRA C configuration.
4. Enable the option **Produce a MISRA C report**.

### Command line syntax

**--misra-c-report**[**=***file*]

### Description

With this option you tell the linker to create a MISRA C Quality Assurance report. This report lists the various modules in the project with the respective MISRA C settings at the time of compilation. If you do not specify a filename, the file *name*`.mcr` is used.

### Example

```
lkm16c --misra-c-report test.obj
```

The linker creates a MISRA C report file `test.mcr`.

### Related information

Compiler option **--misrac**

# –-munch

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Linker** entry and select **Miscellaneous**.

3. Add the option **–-munch** to the **Additional linker options** field.

### Command line syntax

**–-munch**

### Description

With this option you tell the linker to activate the muncher in the pre–locate phase.

### Example

```
lkm16c --munch test.obj
```

The linker activates the muncher in the pre–locate phase while linking the file `test.obj`.

### Related information

–

# –N (––no-rom-copy)

## EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Linker** entry and select **Miscellaneous**.

3. Add the option **–N** to the **Additional linker options** field.

## Command line syntax

**–N**

**––no–rom–copy**

## 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 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.

## Example

```
lkm16c –N test.obj
lkm16c ––no–rom–copy test.obj
```

The linker does not generate a copy table.

## Related information

–

# –-no-rescan

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Linker** entry and select **Libraries**.

3. *Disable* the option **Rescan libraries to solve unresolved externals**.

### Command line syntax

**–-no–rescan**

### Description

When the linker processes a library, it searches for symbol definitions that are referenced by the objects and libraries processed so far. If the library contains a definition for an unresolved reference, the linker extracts the object that contains the definition from the library. The linker processes object files and libraries in the order in which they appear on the command line.

When all objects and libraries are processed the linker checks if there are unresolved symbols left. If so, the default behavior of the linker is to rescan all libraries in the order given on 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.

### Example:

To scan the libraries only once:

```
lkm16c –-no–rescan test.obj a.a b.a
```

The linker resolves all unresolved symbols while scanning the object files and libraries and reports all remaining unresolved symbols after this scan.

### Related information

Linker option **--first-library-first** (Scan libraries in the specified order)

# –-non-romable

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Linker** entry and select **Miscellaneous**.

3. Add the option to the **Additional linker options** field.

### Command line syntax

**–-non–romable**

### Description

With this option you tell the linker that the application is not romable. The linker will locate all ROM sections in RAM. A copy table is generated and is located in RAM. When the application is started, the data sections are re–initialized.

### Example

```
lkm16c --non-romable test.obj
```

The linker locates all ROM sections in RAM.

### Related information

–

# –O (––optimize)

## EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Linker** entry and select **Optimization**.

3. Enable or disable the optimization suboptions.

## Command line syntax

**–O***flags*

**––optimize=***flags*

You can set the following flags:

**c**/**C** (**+**/**–delete–unreferenced–sections**)
Delete unreferenced sections from the output file
(no effect on sources compiled with debug information)

**l**/**L** (**+**/**–first–fit–decreasing**)
Use a 'first fit decreasing' algorithm to locate unrestricted sections in memory.

**t**/**T** (**+**/**–copytable–compression**)
Emit smart restrictions to reduce copy table size

**x**/**X** (**+**/**–delete–duplicate–code**)
Delete duplicate code from the output file

**y**/**Y** (**+**/**–delete–duplicate–data**)
Delete duplicate constant data from the output file

**z**/**Z** (**+**/**–copytable–item–compression**)
Try to compress ROM sections of copy table items

Use the following options for predefined sets of flags:

| | | |
|---|---|---|
| **–O0** | (**––optimize=0**) | No optimization.<br>Alias for: **–OCLTXYZ** |
| **–O1** | (**––optimize=1**) | Normal optimization (default).<br>Alias for: **–OCLtXYZ** |

**–O2** (**––optimize=2**) All optimizations.
Alias for: **–Ocltxyz**

### Default

**–O1**

### Description

With this option you can control the level of optimization. If you do not use this option, **–OCLtXYZ** (**–O1**) is the default.

### Example

The following invocations are equivalent and result all in the default optimizations.

```
lkm16c test.obj
lkm16c -O test.obj
lkm16c -O1 test.obj
lkm16c -OCLtXYZ test.obj

lkm16c --optimize test.obj
lkm16c --optimize=1 test.obj
lkm16c  --optimize=-delete-unreferenced-sections,
   -first-fit-decreasing,+copytable-compression,
   -delete-duplicate-code,-delete-duplicate-data,
   -copytable-item-compression  test.obj
```

### Related information

Section 7.2.3, *Linker Optimizations*, in Chapter *Using the Linker* of the *User's Guide*.

# –o (––output-file)

## EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Linker** entry and select **Miscellaneous**.

3. Add the option **–o** in the **Additional linker options** field.

## Command line syntax

**–o**[*filename*][**:***format*[**:***addr_size*][**,***space_name*]]...

**––output=**[*filename*][**:***format*[**:***addr_size*][**,***space_name*]]...

You can specify the following formats:

| | |
|---|---|
| **ELF** | ELF/DWARF |
| **IEEE** | IEEE–695 |
| **IHEX** | Intel Hex |
| **SREC** | Motorola S–records |

## Description

By default, the linker generates an output file in ELF/DWARF format, named after the first input file with extension `.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 **–o** option multiple times. This useful to generate multiple output formats or to link multiple address spaces. With the first occurrence of the **–o** option you must specify the filename without extension. If you do not specify a filename, or you do not specify the **–o** option at all, the linker uses the default basename `taskn`.

### *IHEX and SREC formats*

If you specify the Intel Hex format or the Motorola S–records format, you can use the argument *addr_size* to specify the size of addresses in bytes (record length). For Intel Hex you can use the values: **1**, **2**, and **4** (default). For Motorola S–records you can specify: **2** (S1 records), **3** (S2 records, default) or **4** bytes (S3 records).

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 `.s` and contains the specified space. (Remember to use the **–o** option multiple times to link multiple address spaces.)

If you do not specify *space_name*, the default address space is emitted. In this case the name of the output file will be *filename_spacename* with the extension `.hex` or `.s`.

Use option **–c** (**––chip–output**) to create Intel Hex or Motorola S–record output files for each chip (suitable for loading into a PROM–programmer).

### Example

To create the output file `myfile.hex` of the address space named `far`:

```
lkm16c test.obj –omyfile:IHEX:2,far
lkm16c test.obj ––output–file=myfile:IHEX:2,far
```

To create the output file `myfile_far.hex` of the default address space:

```
lkm16c test.obj –omyfile:IHEX:2
lkm16c test.obj ––output–file=myfile:IHEX:2
```

### Related information

Linker option **–c** (Generate an output file for each chip)

# –r (––incremental)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Linker** entry and select **Miscellaneous**.

3. Add the option **–r** in the **Additional linker options** field.

### Command line syntax

**–r**

**––incremental**

### Description

Normally the linker links *and* locates the specified object files. With this option you tell the linker to link only the specified files. The linker creates a linker output file `.eln`. 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 `.eln`. The linker will now locate the file.

### Example

In this example, the files `test1.obj`, `test2.obj` and `test3.obj` are incrementally linked:

1. ```
   lkm16c –r test1.obj test2.obj –otest.eln
   lkm16c ––incremental test1.obj test2.obj –otest.eln
   ```

   *`test1.obj` and `test2.obj` are linked*

2. ```
   lkm16c –r test3.obj test.eln
   lkm16c ––incremental test3.obj test.eln
   ```

   *`test3.obj` is linked together with `test.eln`. The file `task1.eln` is created.*

3. ```
   lkm16c task1.eln
   ```

   *`task1.eln` is located*
   ```

### Related information

Section 7.5, *Incremental Linking*, in Chapter *Using the Linker* of the *User's Guide*.

# –S (––strip-debug)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Linker** entry and select **Miscellaneous**.

3. Enable the option **Strip symbolic debug information**.

### Command line syntax

**–S**

**––strip–debug**

### Description

With this option you specify not to include symbolic debug information in the resulting output file.

### Example

```
lkm16c –S test.obj –otest.elf
lkm16c ––strip–debug test.obj ––output=test.elf
```

The linker generates the object file `test.elf` without symbolic debug information.

### Related information

–

# –V (––version)

### EDE

–

### Command line syntax

**–V**

### Description

Display version information. The linker ignores all other options or input files.

### Example

```
lkm16c –V
lkm16c ––version
```

The linker does not link any files but displays the following version information:

```
TASKING M16C linker                          vx.yrz Build 000
Copyright 2002-year Altium BV                Serial# 00000000
```

### Related information

–

# –v (––verbose)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Linker** entry and select **Miscellaneous**.

3. Add the option **–v** to the **Additional linker options** field.

### Command line syntax

**–v**

### Description

With this option you put the linker in *verbose* mode. The linker prints the filenames and the link passes while it processes the files. It also shows which objects are extracted from libraries. With this option you can monitor the current status of the linker.

### Example

```
lkm16c test.obj –lcs –lfps –lrts –v
```

The linker links the file `test.obj` and displays the steps it performs.

### Related information

–

# –w (––no-warnings)

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Linker** entry and select **Warnings**.

3. Enable one of the options **Report all warnings**, **Suppress all warnings**, or **Suppress specific warnings**.

   *If you select **Suppress specific warnings**:*

4. Enter the numbers, separated by commas, of the warnings you want to suppress.

### Command line syntax

**–w**[*nr*[**,***nr*]...]

**––no–warnings**[**=***nr*[**,***nr*]...]

### Description

With this option you can suppresses all warning messages or specific warning messages.

- If you do not specify this option, all warnings are reported.
- If you specify this option but without numbers, all warnings are suppressed.
- If you specify this option with a number, only the specified warnings are suppressed. Separate multiple warnings by commas.

### Example:

To suppress all warnings:

```
lkm16c –w test.obj
lkm16c ––no–warnings test.obj
```

To suppress warnings 135 and 136:

```
lkm16c –w135,136 test.obj
lkm16c ––no–warnings=135,136 test.obj
```

### Related information

Linker option **--warnings-as-errors** (Treat warnings as errors)

# --warnings-as-errors

### EDE

1. From the **Project** menu, select **Project Options...**

   *The Project Options dialog appears.*

2. Expand the **Linker** entry and select **Warnings**.

3. Enable the option **Treat warnings as errors**.

### Command line syntax

**--warnings-as-errors**

### Description

With this option you tell the linker to treat warnings as errors.

When the linker detects an error, it tries to continue the link process and reports other errors and warnings. However, the linker will exit with an exit status not equal zero (!= 0) and will not produce any output files.

### Example

```
lkm16c --warnings-as-errors test.obj
```

When a warning occurs, the linker considers it as an error.

### Related information

Linker option **-w** (Suppress some or all warnings)

## 4.4 CONTROL PROGRAM OPTIONS

The control program **ccm16c** facilitates the invocation of the various components of the M16C toolchain from a single command line. The control program is a command line tool so there are no equivalent options in EDE.

For the linker options in EDE, EDE invokes the linker via the control program. Therefore, it uses the syntax of the control program to pass options and files to the linker. See section 4.3, *Linker Options*, for an overview of the EDE linker options and the corresponding command line linker options.

Some options are interpreted by the control program itself; other options are passed to those programs in the toolchain that accept the option.

### *Recognized input files*

The control program recognizes the following input files:

- Files with a `.cc`, `.cxx` or `.cpp` suffix are interpreted as C++ source programs and are passed to the C++ compiler.
- Files with a `.c` suffix are interpreted as C source programs and are passed to the compiler.
- Files with a `.asm` suffix are interpreted as hand–written assembly source files which have to be passed to the assembler.
- Files with a `.src` suffix are interpreted as compiled assembly source files. They are directly passed to the assembler.
- Files with a `.a` 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 `.eln` suffix are interpreted as linked object files and are passed to the locating phase of the linker. The linker accepts only one `.eln` file in the invocation.
- An argument with a `.lsl` suffix is interpreted as a linker script file and is passed to the linker.

Normally, the control program tries to compile, assemble, link and locate all source files to absolute object files. There are however, options to suppress the assembler, link or locate stage.

# –? (––help)

### Command line syntax

**–?**[**options**]

**––help**[**=options**]

### Description

Displays an overview of all command line options. When you specify the suboption **options**, you receive extended information.

### Example

The following invocations all display a list of the available command line options:

```
ccm16c –?
ccm16c ––help
ccm16c
```

### Related information


–

# –A (––language)

## Command line syntax

**–A**[*flags*]

**––language**[**=***flags*]

You can set the following flags:

| | | |
|---|---|---|
| **p**/**P** | (**+**/**–comments**) | Allow C++ style comments in ISO C90 |
| **x**/**X** | (**+**/**–strings**) | Relaxed const check for string literals |

## Default

**–Apx**

## Description

With this option you control the language extensions the compiler can accept. Default the C compiler allows all language extensions.

**–A** (**––language**) is the equivalent of **–APX** and disables all language extensions.

With **–Ap** you tell the compiler to allow C++ style comments (//) in ISO C90 mode (option **–c90**). In ISO C99 mode this style of comments is always accepted.

With **–Ax** you tell the compiler not to check for assignments of a constant string to a non–constant string pointer. With this option the following example does not produces a warning:

```
char *p;
void main( void ) { p = ”hello”; }
```

## Example

```
ccm16c –APx test.c
ccm16c ––language=–comments,+strings test.c
```

The control program calls the compiler in such a way that it accepts assignments of a constant string to a non–constant string pointer but ignores C++ style comments.

### Related information

Compiler option **–A** (Control language extensions)

# –address-size

### Command line syntax

**–address–size=***addr_size*

### Description

If you specify IHEX or SREC with the control option **–format**, you can additionally specify the record length and the address space to be emitted in the output files.

With this option you can specify the size of addresses in bytes (record length). For Intel Hex you can use the values: **1**, **2**, and **4** (default). For Motorola S–records you can specify: **2** (S1 records), **3** (S2 records, default) or **4** bytes (S3 records).

If you do not specify *addr_size*, the default address size is generated.

### Example

To create the SREC file `test.s` with S1 records, type:

```
ccm16c –format=SREC –address–size=2
```

### Related information

Control program option **–format** (Set linker output format)
Control program option **–space** (Set linker output space name)

Linker option **–o** (Specify an output object file)

# –C (––cpu)

## Command line syntax

**–C***cpu*

**––cpu=***type*

## Description

With this option you define the target processor for which you create your application.

Based on the target processor, the compiler includes a *special function register file* `reg`*`cpu`*`.sfr`. This is an include file written in C syntax which is shared by the compiler, assembler and debugger. Once the compiler reads an SFR file you can reference the special function registers (SFR) and bits within an SFR using symbols defined in the SFR file.

## Example

To build the file `test.c` for the `m30624a` processor and use the SFR file `regm30624a.sfr`:

```
cm16c –Cm30624a test.c
cm16c ––cpu=m30624a test.c
```

## Related information

Compiler option **–C** (Select the CPU type)

Assembler option **–C** (Select CPU)

# –cc/–cs/–co/–cl (––create)

## Command line syntax

**–cc**
**––create=c**

**–cs**
**––create=assembly**

**–co**
**––create=object**

**–cl**
**––create=relocatable**

## 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.

**–cc** Stop after C++ files are compiled to intermediate C files (`.ic`)
**–cs** Stop after C files are compiled to assembly (`.src`)
**–co** Stop after the files are assembled to object files (`.obj`)
**–cl** Stop after the files are linked to a linker object file (`.eln`)

## Example

To generate the object file `test.obj`:

```
ccm16c –co test.c
ccm16c ––create=object test.c
```

The control program stops after the file is assembled. It does not link nor locate the generated output.

## Related information

–

# --check

### Command line syntax

**--check**

### Description

With this option you can check the source code for syntax errors, without generating code. This saves time in developing your application.

The compiler/assembler reports any warnings and/or errors.

### Example

To check for syntax errors, without generating code:

```
ccm16c --check test.c
```

### Related information

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

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

# –D (––define)

## Command line syntax

**–D***macro_name*[**=***macro_definition*]

**––define=***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. On the command line, use the option **–D** multiple times. If the command line exceeds the length limit of the operating system, you can define the macros in an option file which you then must specify to the control program with the option **–f** *file*.

Defining macros with this option (instead of in the C source) is, for example, useful to compile or assemble conditional source as shown in the example below.

The control program passes the option **–D** (**––define**) to the compiler and the assembler.

## Example

Consider the following C program with conditional code to compile a demo program and a real program:

```
void main( void )
{
#if DEMO == 1
    demo_func();  /* compile for the demo program */
#else
    real_func();  /* compile for the real program */
#endif
}
```

You can now use a macro definition to set the DEMO flag. With the control program this looks as follows:

```
ccm16c –DDEMO test.c
ccm16c –DDEMO=1 test.c
```

```
ccm16c --define=DEMO test.c
ccm16c --define=DEMO=1 test.c
```

Note that all four 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.

```
ccm16c -D"MAX(A,B)=((A) > (B) ? (A) : (B))"
ccm16c --define="MAX(A,B)=((A) > (B) ? (A) : (B))"
```

### Related information

Control program option **–U** (Undefine preprocessor macro)
Control program option **–f** (Read options from file)

# –d (––lsl-file)

## Command line syntax

**–d***file*

**––lsl–file=***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 and derivative definition describe the core's hardware architecture and its internal memory.
- the board specification describes the physical memory available in the system.
- the section layout definition describes how to locate sections in memory.

With this option you specify a linker script file via the control program to the linker. If you do not specify this option, the linker does not use a script file. You can specify the existing file `m16c.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.

## Example

To read linker script file information from file `mylslfile.lsl`:

```
ccm16c –dmylslfile.lsl test.obj
ccm16c ––lsl–file=mylslfile.lsl test.obj
```

## Related information

Section 7.6, *Controlling the Linker with a Script*, in the User's Guide

# --diag

## Command line syntax

**--diag=**[*format*:]{**all** | *nr*,...]

## Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to stdout (normally your screen) and in the format you specify. You can specify the following formats: **html**, **rtf** or **text** (default). To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas.

With this option the control program does not process any files.

## Example

To display an explanation of message number 103 , enter:

```
ccm16c --diag=103
```

This results in message 103 with explanation.

To write an explanation of all errors and warnings in HTML format to file `ccerrors.html`, enter:

```
ccm16c --diag=html:all > ccerrors.html
```

## Related information


–

# –E (––preprocess)

## Command line syntax

**–E**[*flags*]

**––preprocess=**[*flags*]

You can set the following flags:

| | | |
|---|---|---|
| **c**/**C** | (**+**/**–comments**) | Keep comments |
| **p**/**P** | (**+**/**–noline**) | Strip #line source position info |

## Description

With this option you tell the control program to preprocess the C source.

The compiler sends the preprocessed file to stdout. To capture the information in a file, specify an output file with the option **–o**.

With **–Ec** you tell the preprocessor to keep the comments from the C source file in the preprocessed output.

With **–Ep** 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 more orderly to read.

## Example

```
ccm16c –EcP test.c –o test.pre
ccm16c ––preprocess +comments,–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 the line source position information is not stripped from the output file.

## Related information

–

# –-error-file

### Command line syntax

**–-error-file**[**=***file*]

### Description

With this option the control program 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:

```
ccm16c –-error-file=errors.err test.c
```

### Related information

Control program option **–-warnings–as–errors** (Warnings as errors)

# --exceptions

## Command line syntax

**--exceptions**

## Description

With this option you enable support for exception handling in the C++ compiler.

## Example

To enable exception handling, enter:

```
ccm16c --exceptions test.cc
```

## Related information


–

# –F (––no-double)

## Command line syntax

**–F**
**––no–double**

## Description

With this option you tell the control program 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

```
ccm16c –F test.c
ccm16c ––no–double test.c
```

The file `test.c` is processed where variables of the type double are treated as float in the compilation phase.

## Related information


–

# –f (––option-file)

## Command line syntax

**–f** *file*

**––option–file=***file*

## Description

Instead of typing all options on the command line, you can create a option file which contains all options and file you want to specify. With this option you specify the option file to the control program.

Use an option file when the length of the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option **–f** multiple times.

### *Format of an option file*

- Multiple command line 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:

```
-g
-k
test.c
```

Specify the option file to the control program:

```
ccm16c -f myoptions
ccm16c --option-file=myoptions
```

This is equivalent to the following command line:

```
ccm16c -g -k test.c
```

# --force-c

## Command line syntax

**--force-c**

## Description

With this option you tell the control program to treat all `.cc` files as C files instead of C++ files. This means that the control program does not call the C++ compiler and forces the linker to link C libraries.

## Example

```
ccm16c --force-c test.cc
```

The C++ file `test.cc` is considered to be a normal C file.

## Related information

Control program option **--force-c++** (Force C++ compilation and linking)

# –-force-c++

### Command line syntax

**–-force–c++**

### Description

With this option you tell the control program to treat all `.c` files as C++ files instead of C files. This means that the control program calls the C++ compiler prior to the C compiler and forces the linker to link C++ libraries.

### Example

```
ccm16c –-force–c++ test.c
```

The file `test.c` is considered to be a C++ file.

### Related information

Control program option **–-force–c** (Treat C++ files as C files)

# –-force-munch

### Command line syntax

**–-force–munch**

### Description

With this option you force the control program to activate the muncher in the pre–locate phase.

### Example

To force the mucher phase in the pre–locate phase, type:

```
ccm16c –-force–munch test.cc
```

### Related information

–

# –-force-prelink

### Command line syntax

**–-force–prelink**

### Description

With this option you force the control program to invoke the C++ pre–linker.

### Example

```
ccm16c –-force–prelink test.cc
```

The control program always invokes the C++ pre–linker when generating `test.elf`.

### Related information

–

# –-format

## Command line syntax

**–-format=***format*

You can specify the following formats:

| | |
|---|---|
| **ELF** | ELF/DWARF |
| **IEEE** | IEEE–695 |
| **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 CrossView Pro debugger.

If you choose IHEX or SREC, you can additionally specify the address size of the chosen format (option **–-address–size**) and the address space to be emitted (option **–-space**).

## Example

To generate an ELF/DWARF output file:

```
ccm16c --format=ELF test1.c test2.c --output=test.elf
```

## Related information

Control program option **–-address–size** (For linker IHEX./SREC files)
Control program option **–-space** (Set linker output space name)

Linker option **–o** (output file)
Linker option **–c** (generate hex file)

Section 6.1, *ELF/DWARF Object Format*, in Chapter *Object File Formats*.

# –-fp-trap

## Command line syntax

**–-fp–trap**

## Description

By default the control program uses the non–trapping floating point library (`libfp.a`). With this option you tell the control program to use the trapping floating point library (`libfpt.a`).

If you use the trapping floating point library, exceptional floating point cases are intercepted and can be handled separately by an application defined exception handler. Using this library decreases the execution speed of your application.

## Example

```
ccm16c –-fp–trap test.c
```

Link the trapping floating point library when generating the object file `test.elf`.

## Related information

–

# –g (––debug-info)

## Command line syntax

**–g**

**––debug–info**

## Description

With this option you tell the control program to include debug information in the generated object file.

## Example

```
ccm16c –g test.c
ccm16c ––debug–info test.c
```

The control program includes symbolic debug information in the generated object file `test.elf`.

## Related information

–

# –I (––include-directory)

## Command line syntax

**–I***path*

**––include–directory=***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.

## 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:

```
ccm16c –Imyinclude test.c
ccm16c ––include–directory=myinclude
```

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

Compiler option **–I** (Add directory to include file search path)
Compiler option **–H** (Include file at the start of a compilation)

# --instantiate

### Command line syntax

**--instantiate=***mode*

### Description

Normally, when a file is compiled, no template entities are instantiated (except those assigned to the file by automatic instantiation). The overall instantiation mode can, however, be changed with this option. You can specify the following modes:

**none** Do not automatically create instantiations of any template entities. This is the default. It is also the usually appropriate mode when automatic instantiation is done.

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

**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.

### Example

To specify instantiation mode `used`, type

```
ccm16c --instantiate=used test.cc
```

### Related information

–

# –-instantiation-dir

### Command line syntax

**–-instantiation-dir=***dir*

### Description

With this option the C++ compiler generates additional files for template instantiations in the specified directory.

If you do not specify this option, files are created in the current directory.

If you specify the control program option **–-no-one-instantiation-per-object**, this option remains without effect.

### Example

To specify the directory for instantiation files, type

```
ccm16c --instantiation-dir=instant test.cc
```

### Related information

Control program option **–-no-one-instantiation-per-object**

# --instantiation-file

### Command line syntax

**--instantiation-file=***file*

### Description

With this option the C++ compiler generates a list of all generated template instantiation files and writes it to the specified *file*. You can use this file for example to use as an option file for the archiver **arm16c**.

### Example

To create a file with a list of all generated instantiation files, type

```
ccm16c --instantiation-file=instlist.ii test.cc
```

### Related information

 –

# --iso

### Command line syntax

**--iso={90|99}**

### Description

With this option you specify to the control program against which ISO standard it should check your C source. C90 is also referred to as the ”ANSI C standard”. C99 refers to the newer ISO/IEC 9899:1999 (E) standard and is the default.

Independant of the chosen ISO standard, the control program always links libraries with C99 support.

### Example

To compile the file `test.c` conform the ISO C90 standard:

```
ccm16c --iso=90 test.c
```

### Related information

Compiler option **-c** (ISO C standard)

# –k (––keep-output-files)

## Command line syntax

**–k**

**––keep–output–files**

## 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.

## Example

```
ccm16c –k test.c
ccm16c ––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


–

# –L (––library-directory / ––ignore-default-library-path)

## Command line syntax

**–L***path*
**––library–directory=***path*

**–L**
**––ignore–default–library–path**

## Description

With this option you can specify the path(s) where your system libraries, specified with the **–l** option, are located. If you want to specify multiple paths, use the option **–L** for each separate path.

The default path is `$(PRODDIR)\lib\m16c`.

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 LIBM16C. 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 **–l** option is:

1. The path that is specified with the **–L** option.
2. The path that is specified in the environment variable LIBM16C when the product was installed.
3. The default directory `$(PRODDIR)\lib\m16c`.

## Example

Suppose you call the control program as follows:

```
ccm16c test.c –Lc:\mylibs –lcs
ccm16c test.c ––library–directory=c:\mylibs –lcs
```

First the linker looks in the directory `c:\mylibs` for library `libcs.a` (this option).

If it does not find the requested libraries, it looks in the directory that is set with the environment variable LIBM16C.

Then the linker looks in the default directory `$(PRODDIR)\lib\m16c` for libraries.

### Related information

Linker option **–l** (Search also in system library lib*name*)

# –l (––library)

## Command line syntax

**–l***name*

**––library=***name*

## Description

With this option you tell the linker via the control program to search also in system library `lib`*`name`*`.a`, where *name* is a string. The linker first searches for system libraries in any directories specified with **–L***path*, then in the directories specified with the environment variable LIBM16C, unless you used the option **–L** without a directory.

## Example

To search in the system library `libfps.a` (floating–point library):

```
ccm16c test.obj mylib.a –lfps
ccm16c test.obj mylib.a ––library=fps
```

The linker links the file `test.obj` and first looks in `mylib.a` (in the current directory only), then in the system library `libfps.a` to resolve unresolved symbols.

## Related information

Control program option **–L** (Add library directory)

Section 7.4, *Linking with Libraries*, in the User's Guide

# --list-object-files

### Command line syntax

**--list-object-files**

### Description

With this option the list of object files that are handled by the prelinker, is displayed at `stdout`. The list is shown when it is changed by the prelinker.

### Example

To show the list of object files handled by the prelinker, enter:

```
ccm16c --list-object-files test.cc
```

### Related information

 –

# –M (––model)

## Command line syntax

**–M{l | m | s}**

**––model={large | medium | small}**

## Description

By default the **cm16c** compiler uses the small memory model. This model generates the most efficient code. You can use the option **–M** to specify another memory model to the control program.

The table below illustrates the meaning of each data model:

| Model | Data | Constants | Pointers |
|---|---|---|---|
| Small | __near: in first 64 kB | __near | __near |
| Medium | __near: in first 64 kB | __paged | __paged |
| Large | __far: anywhere in 1 MB | __far | __far |

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:

small model 's'
medium model 'm'
large model 'l'

## Example

To compile the file `test.c` for the large memory model:

```
ccm16c -Ml test.c
ccm16c --model=large test.c
```

## Related information

Compiler option **–M** (Select memory model)

# –n (––dry-run)

## Command line syntax

**–n**
**––dry–run**

## Description

With this option you put the control program *verbose* mode. The control program prints the invocations of the tools it would use to process the files.

## Example

To see how the control program will invoke the tools it needs to process the file `test.c`:

```
ccm16c –n test.c
ccm16c ––dry–run test.c
```

The control program only displays the invocations of the tools it would use to create the final object file but does not actually perform the steps.

## Related information

Control program option **–v** (Verbose output)

# –-no-auto-instantiation

## Command line syntax

**–-no–auto–instantiation**

## Description

Default, the c++ compiler automatically instantiates templates. With this option automatic instantiation of templates is disabled.

## Example

To disable automatic instantiation, type

```
ccm16c –-no–auto–instantiation test.cc
```

## Related information

For an extensive description of automatic insantiation, refer to section 2.6.1, *Automatic Instantiation*, in the *M16C C++ Compiler User's Guide*.

# –-no-default-libraries

## Command line syntax

**–-no–default–libraries**

## Description

Default the control program specifies the standard C libraries 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 **–l***library_name*. The control program recognizes the option **–l** as an option for the linker.

## Example

```
ccm16c –-no–default–libraries test.c
```

The control program does not specify any libraries to the linker. In normal cases this would result in unresoved externals.

To specify your own libraries (`libmy.a`) and avoid unresolved externals:

```
ccm16c –-no–default–libraries –lmy test.c
```

## Related information

Linker option **–l** (Search also in system library `libx.a`)

# --no-map-file

## Command line syntax

**--no-map-file**

## Description

By default the control program generates a linker map file (`.map`).

A linker map file is a text file that shows how the linker has mapped the sections and symbols from the various object files (`.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.

## Example

To prevent the generation of the linker map file `test.map`:

```
ccm16c --no-map-file test.c
```

## Related information

Linker option **-M** (Generate map file)

# –-no-one-instantiation-per-object

### Command line syntax

**–-no–one–instantiation–per–object**

### Description

With this option, the C++ compiler writes template instantiations into a single object file. If you do not specify this option, the C++ compiler creates multiple files. In that case you can specify a directory for those files with the control program option **–-instantiation–dir**.

### Example

To create a file with a list of all generated instantiation files, type

```
ccm16c –-no–one–instantiation–per–object test.cc
```

### Related information

Control program option **–-instantiation–dir**

# –o (––output)

## Command line syntax

**–o***file*

**––output=***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.

## Example

```
ccm16c 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`:

```
ccm16c –oresult.elf test.c prog.c
ccm16c ––output=result.elf test.c prog.c
```

## Related information

–

# --prelink-copy-if-non-local

### Command line syntax

**--prelink-copy-if-non-local**

### Description

If a file must be recompiled and it is not in the current directory, with this option the C++ prelinker copies the prelink file (`.ii`) to the current directory and rewrites that .ii file so it can find its associated `.cc` file. As a result, the `.cc` file is recompiled in the current directory.

With this option you prevent that previously compiled files are overwritten during recompilation.

### Example

To copy all files for recompilation to the current directory:

```
ccm16c --prelink-copy-if-non-local test.cc
```

### Related information


–

# --prelink-local-only

### Command line syntax

**--prelink-local-only**

### Description

With this option the C++ prelinker ignores all files that are outside the current directory.

### Example

To prelink only files in the current directory:

```
ccm16c --prelink-local-only test.cc
```

### Related information


–

# --prelink-remove-instantiation-flags

### Command line syntax

**--prelink-remove-instantiation-flags**

### Description

With this option the C++ prelinker removes all instantiation flags from the generated object files.

### Example

To remove instantiation flags from the generated object files:

```
ccm16c --prelink-remove-instantiation-flags test.cc
```

### Related information


–

# --r8c

## Command line syntax

**--r8c**

## Description

By default, the control program generates code for the M16C/60 core. With this option you tell the control program to generate files for R8C/tiny core. You must use this option always (and only then) when you select an R8C target with the control program option **–C***cpu*.

When you use this option:

- Other vector addresses are generated with the `__interrupt_fixed` keyword
- `__far` and `__paged` memory type qualifiers are interpreted as `__near`
- because code resides in near memory, pointers to functions are 16 bit (instead of 32 bit)
- Code sections are always placed in near memory.

To avoid conflicts, make sure you specify this option also to the assembler.

## Example

```
ccm16c --r8c --Cr8c10 test.c
```

## Related information

Control program option **–C***cpu* (Select target CPU)

# --show-c++-warnings

### Command line syntax

**--show-c++-warnings**

### Description

The C++ compiler may generate a compiled C++ file (`.ic`) that causes warnings during compilation or assembling. With this option you tell the control program to show these warnings. Default C++ warnings are suppressed.

### Example

```
ccm16c --show-c++-warnings test.cc
```

The control program calls the C++ compiler which generates the C file (`test.ic`). If this file causes warnings during compilation or assembling, these warnings are shown.

### Related information


–

# --space

## Command line syntax

**--space=***space_name*

## Description

If you specify IHEX or SREC with the control option **--format**, you can additionally specify the record length and the address space to be emitted in the output files.

With this option you can specify which address space must be emitted. 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 `.s`.

If you do not specify *space_name*, the default address space is emitted. In this case the name of the output file will be *filename_spacename* with the extension `.hex` or `.s`.

## Example

To create the IHEX file `test.hex`, type:

```
ccm16c --format=IHEX --space=far test.c
```

If the specified memory space does not exist, the control program emits the default space name and reflects this in the output file name.

## Related information

Control program option **--format** (Set linker output format)

Linker option **-o** (Specify an output object file)

# --static

### Command line syntax

**--static**

### Description

This option is directly passed to the compiler.

With this option, the compiler treats external definitions at file scope (except for `main`) as if they were declared `static`. As a result, unused functions will be eliminated, and the alias checking algorithm assumes that objects with static storage cannot be referenced from functions outside the current module.

This option only makes sense when you specify all modules of an application on the command line.

### Example

```
ccm16c --static module1.c module2.c module3.c
```

### Related information


–

# –t (––keep-temporary-files)

## Command line syntax

**–t**

**––keep–temporary–files**

## Description

By default, the control program removes intermediate files like the `.src` file (result of the compiler phase) and the `.eln` file (result of the linking phase).

With this option you tell the control program to keep temporary files it generates during the creation of the absolute object file.

## Example

To keep all temporary files:

```
ccm16c –t test.c
ccm16c ––keep–temporary–files test.c
```

The control program keeps all intermediate files it generates while creating the absolute object file `test.elf`.

## Related information

–

# –U (––undefine)

## Command line syntax

**–U***macro_name*
**––undefine=***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.

However, 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 **–U** (**––undefine**) to the compiler.

## Example

To undefine the predefined macro __TASKING__:

```
ccm16c -U__TASKING__ test.c
ccm16c --undefine=__TASKING__ test.c
```

## Related information

Control Pogram option **–D** (Define preprocessor macro)

# -V (--version)

## Command line syntax

**-V**
**--version**

## Description

Display version information. The control program ignores all other options or input files.

## Example

```
ccm16c -V
ccm16c --version
```

The control program does not call any tools but displays the following version information:

```
TASKING M16C control program                vx.yrz Build nnn
Copyright years Altium BV                   Serial# 00000000
```

## Related information


–

# –v (––verbose)

### Command line syntax

**–v**
**––verbose**

### Description

With this option you put the control program in *verbose* mode. With the option **–v** the control program performs it tasks while it prints the steps it performs to `stdout`.

### Example

```
ccm16c –v test.c
ccm16c ––verbose test.c
```

The control program processes the file `test.c` and displays the invocations of the tools it uses to create the final object file

### Related information

Control program option **–n** (Verbose output and suppress execution)

# –W*tool* (––pass)

## Command line syntax

| | | |
|---|---|---|
| **–Wcp***option* | **––pass–c++=***option* | Pass option directly to the C++ compiler |
| **–Wc***option* | **––pass–c=***option* | Pass option directly to the C compiler |
| **–Wa***option* | **––pass–assembler=***option* | Pass option directly to the assembler |
| **–Wpl***option* | **––pass–prelinker=***option* | Pass option directly to the C++ prelinker |
| **–Wl***option* | **––pass–linker=***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 the option itself, but specifies it directly to the tool which the control program calls.

## Example

```
ccm16c –Wl–r test.c
```

The control program does not use the option **–r** but calls the linker with the option **–r** (`lkm16c –r`).

## Related information

–

# –w (no-warnings)

### Command line syntax

**–w**[*nr*]
**––no–warnings**[**=***nr*]

### Description

With this option suppresses all warning messages or a specific warning. If you do not specify this option, all warnings are reported.

### Example

To suppress all warnings:

```
ccm16c –w test.c
ccm16c ––no–warnings test.c
```

To suppress warnings 100:

```
ccm16c –w100 test.c
ccm16c ––no–warnings=100 test.c
```

### Related information

Control program option **––warnings–as–errors** (Warnings as errors)

# --warnings-as-errors

### Command line syntax

**--warnings-as-errors**

### Description

With this option you tell the control program to treat warnings as errors.

### Example

```
ccm16c --warnings-as-errors test.c
```

When a warning occurs, the control program considers it as an error.

### Related information

Control program option **-w** (Suppress all warnings)

## 4.5 MAKE UTILITY OPTIONS

When you build a project in EDE, EDE generates a makefile and uses the graphical make utility **wmk** to build all your files. However, you can also use the make utility **mkm16c** from the command line to build your project.

The invocation syntax is:

```
mkm16c [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 EDE.

# Defining Macros

### Command line syntax

*macro*=*definition*

### Description

With this argument you can define a macro and specify it to the make utility.

A macro definition remains in existence during the execution of the makefile, even when the makefile recursively calls the make utility again. In the recursive call, the macro acts as an environment variable. This means that it is overruled by definitions in the recursive call. Use the option **–e** to prevent this.

You can specify as many macros as you like. If the command line exceeds the limit of the operating system, you can define the macros in an *option file* which you then must specify to the compiler 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.eln : demo.obj
             lkm16c demo.obj main.obj –lc –lfp –lrt
else
  real.eln : real.obj
             lkm16c real.obj main.obj –lc –lfp –lrt
endif

real.elf   : real.eln
             lkm16c –FELF –oreal.elf real.eln
```

You can now use a macro definition to set the `DEMO` flag:

```
mkm16c real.elf DEMO=1
```

In both cases the absolute obect file `real.elf` is created but depending on the DEMO flag it is linked with `demo.obj` or with `real.eln`.

### Related information

Make utility option **–e** (Environment variables override macro definitions)
Make utility option **–m** (Name of invocation file)

# –?

## Command line syntax

```
–?
```

## Description

Displays an overview of all command line options.

## Example

To display a list of the available command line options:

```
mkm16c –?
```

## Related information

–

# –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

```
mkm16c –a
```

Rebuilds all your files, regardless of whether they are out of date or not.

## Related information

–

# –c

## Command line syntax

**–c**

## Description

EDE uses this option for the graphical version of make when you create sub–projects. In this case make calls another instance of make 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

The following command runs the make utility as a child process:

```
mkm16c -c
```

## Related information

Make utility option **–err** (Redirect error message to file)

# –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 **mkm16c**.

With the option **–DD** not only the lines of the makefile are printed but also the lines of the `mkm16c.mk` file (implicit rules).

## Example

```
mkm16c –D
```

Each line of the makefile that is read by the make utility is printed to standard output (usually your screen).

## Related information

–

# –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

```
mkm16c –d
```

Shows which files are out of date and rebuilds them.

### Related information


–

# –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

```
mkm16c –e
```

The make utility uses the settings of the environment variables regardless of macro definitions.

## Related information

–

# –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

```
mkm16c –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)

# –f

## Command line syntax

**–f** *my_makefile*

## Description

Default the make utility uses the file **makefile** to build your files.

With this option you tell the make utility to use the specified file instead of the file **makefile**. Multiple **–f** options act as if all the makefiles were concatenated in a left–to–right order.

## Example

```
mkm16c mymake
```

The make utility uses the file **mymake** to build your files.

## Related information

–

# –G

## Command line syntax

**–G** *path*

## Description

Normally you must call the make utility **mkm16c** 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 as follows:

```
mkm16c –G ..\myfiles
```

## Related information

–

# –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

```
mkm16c -i
```

The make utility exits without an error code, even when an error occurs.

## Related information

–

# –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 variable is not specified.

### Example

```
mkm16c –K
```

The make utility preserves all temporary files.

### Related information

Section 1.3.2, *Configuring the Command Line Environment*, in Chapter *Software Installation* of the *User's Guide*.

# –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

```
mkm16c -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**)

# –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 length of the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option **–m** multiple times.

### *Format of an option file*

- Multiple command line 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:

```
mkm16c -m myoptions
```

This is equivalent to the following command line:

```
mkm16c -k -err errors.txt test.elf
```

### Related information


–

# –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

```
mkm16c –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)

# –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

```
mkm16c –p
```

The make utility never removes target dependency files.

## Related information


–

# –q

### Command line syntax

**–q**

### Description

With this option the make utility does not perform any tasks but only returns an error 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

```
mkm16c –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

–

# –r

## Command line syntax

**–r**

## Description

When you call the make utility, it first reads the implicit rules from the file `mkm16c.mk`, then it reads the makefile with the rules to build your files. (The file `mkm16c.mk` is located in the `\etc` directory of the toolchain.)

With this option you tell the make utility *not* to read `mkm16c.mk` and to rely fully on the make rules in the makefile.

## Example

```
mkm16c –r
```

The make utility does not read the implicit make rules in `mkm16c.mk`.

## Related information

–

# –S

## Command line syntax

**–S**

## Description

With this option you cancel the effect of the option **–k**. This is never necessary except in a recursive make where the option **–k** might be inherited from the top–level make via MAKEFLAGS or if you set the option **–k** in the environment variable MAKEFLAGS.

## Example

```
mkm16c –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 **mkm16c** in the makefile.

## Related information

Make utility option **–k** (On error, abandon the work for the current target only)

# –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

```
mkm16c –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)

# –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

```
mkm16c –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

–

# –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

```
mkm16c –time
```

The make utility displays the current date and time and updates out–of–date files.

## Related information

–

# -V

### Command line syntax

**-V**

### Description

Display version information. The make utility ignores all other options or input files.

### Example

```
mkm16c -V
```

The make utility does not perform any tasks but displays the following version information:

```
TASKING M16C program builder    vx.yrz Build nnn
Copyright 2003-year Altium BV   Serial# 00000000
```

### Related information


-

# –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

```
mkm16c –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


–

# –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.

## Example

```
mkm16c –w
```

The make utility sends messages to standard output instead of standard error.

## Related information

–

# –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. EDE uses this option for the graphical version of make.

## Example

```
mkm16c –x
```

If errors occur, the make utility gives extended information.

## Related information


–

## 4.6 ARCHIVER OPTIONS

The archiver and library maintainer **arm16c** 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:

```
arm16c 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.

The archiver is a command line tool so there are no equivalent options in EDE.

| Description | Option | Suboption |
|---|---|---|
| **Main functions** | | |
| 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** | |
| **Miscellaneous** | | |
| Display options | **–?** | |
| Display version header | **–V** | |
| Read options from *file* | **–f** *file* | |
| Suppress warnings above level *n* | **–w***n* | |

*Table 4–1: Overview of archiver options and suboptions*

# –?

## Command line syntax

**–?**

## Description

Displays an overview of all command line options.

## Example

The following invocations display a list of the available command line options:

```
arm16c –?
arm16c
```

## Related information


–

# -d

## Command line syntax

**–d** [**–v**]

## Description

Delete the specified object modules from a library. With the suboption **–v** the archiver shows which files are removed.

**–v** Verbose: the archiver shows which files are removed.

## Example

```
arm16c –d lib.a obj1.obj obj2.obj
```

The archiver deletes `obj1.obj` and `obj2.obj` from the library `lib.a`.

```
arm16c –d –v lib.a obj1.obj obj2.obj
```

The archiver deletes `obj1.obj` and `obj2.obj` from the library `lib.a` and displays which files are removed.

## Related information


–

# –f

## Command line syntax

**–f** *file*

## Description

Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the archiver **arm16c**.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

Option files can also be generated on the fly, for example by the make utility. You can specify the option **–f** multiple times.

### *Format of an option file*

- Multiple arguments on one line in the option file are allowed.
- To include whitespace in an argument, surround the argument with single or double quotes.
- If you want to use single quotes as part of the argument, surround the argument by double quotes and 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 lib.lib obj1.obj
-w5
```

Specify the option file to the archiver:

```
arm16c –f myoptions
```

This is equivalent to the following command line:

```
arm16c –x lib.lib obj1.obj –w5
```

### Related information


–

# –m

## Command line syntax

**–m** [**–a** *posname*] [**–b** *posname*]

## Description

Move the specified object modules to another position in the library.

The ordering of members in a library can make a difference in how programs are linked if a symbol is defined in more than one member.

Default, the specified members are moved to the end of the archive. Use the suboptions **–a** or **–b** to move them to a specified place instead.

**–a** *posname* Move the specified object module(s) after the existing module *posname*.

**–b** *posname* Move the specified object module(s) before the existing module *posname*.

## Example

Suppose the library `lib.a` contains the following objects (see option **–t**):

```
obj1.obj
obj2.obj
obj3.obj
```

To move `obj1.obj` to the end of `lib.a`:

```
arm16c –m lib.a obj1.obj
```

To move `obj3.obj` just before `obj2.obj`:

```
arm16c –m –b obj3.obj lib.a obj2.obj
```

The library `lib.a` after these two invocations now looks like:

```
obj3.obj
obj2.obj
obj1.obj
```

## Related information

Archiver option **–t** (Print library contents)

# –p

### Command line syntax

**–p**

### Description

Print the specified object module(s) in the library to standard output.

This option is only useful when you redirect or pipe the output to other files or tools that serve your own purposes. Normally you do not need this option.

### Example

```
arm16c –p lib.a 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 nam.

### Related information

–

# –r

## Command line syntax

**–r** [**–a** *posname*] [**–b** *posname*] [**–c**] [**–u**] [**–v**]

## Description

You can use the option **–r** for several purposes:

- Adding new objects to the library
- Replacing objects in the library with the same object of a newer date
- Creating a new library

The option **–r** normally *adds* a new module to the library. However, if the library already contains a module with the specified name, the existing module is *replaced*. If you specify a library that does not exist, the 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 to a specified place instead.

| | |
|---|---|
| **–a** *posname* | Add the specified object module(s) after the existing module *posname*. |
| **–b** *posname* | Add the specified object module(s) before the existing module *posname*. |
| **–c** | Create a new library without checking whether it already exists. If the library already exists, it is overwritten. |
| **–u** | Insert the specified object module only if it is newer than the module in the library. |
| **–v** | Verbose: the archiver shows which files are removed. |

The suboptions **–a** or **–b** have no effect when an object is added to the library.

### Examples

Suppose the library `lib.a` contains the following objects (see option **–t**):

```
obj1.obj
```

To add `obj2.obj` to the end of `lib.a`:

```
arm16c –r lib.a obj2.obj
```

To insert `obj3.obj` just before `obj2.obj`:

```
arm16c –r –b obj2.obj lib.a obj3.obj
```

The library `lib.a` 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:

```
arm16c –r obj1.obj newlib.a
```

The archiver creates the library `newlib.a` and adds the object `obj1.obj` to it.

To *create a new library file and overwrite an existing library*, add an object file and specify an existing library with the supoption **–c**:

```
arm16c –r –c obj1.obj lib.a
```

The archiver overwrites the library `lib.a` and adds the object `obj1.obj` to it. The new library `lib.a` only contains `obj1.obj`.

### Related information

Archiver option **–t** (Print library contents)

# –t

## Command line syntax

**–t** [**–s0**|**–s1**]

## Description

Print a table of contents of the library to standard out. With the suboption **–s** you the archiver displays all symbols per object file.

**–s0** Displays per object the library in which it resides, the name of the object itself and all symbols in the object.

**–s1** Displays only the symbols of all object files in the library.

## Example

```
arm16c –t lib.a
```

The archiver prints a list of all object modules in the library `lib.a`.

```
arm16c –t –s0 lib.a
```

The archiver prints per object all symbols in the library. This looks like:

```
prolog.obj
   symbols:
lib.a:prolog.obj:___Qabi_callee_save
lib.a:prolog.obj:___Qabi_callee_restore
div16.obj
   symbols:
lib.a:div16.obj:___udiv16
lib.a:div16.obj:___div16
lib.a:div16.obj:___urem16
lib.a:div16.obj:___rem16
```

## Related information

 –

# –V

## Command line syntax

**–V**

## Description

Display version information. The archiver ignores all other options or input files.

## Example

```
arm16c –V
```

The archiver does not perform any tasks but displays the following version information:

```
TASKING M16C ELF archiver       vx.yrz Build nnn
Copyright 2003-year Altium BV   Serial# 00000000
```

## Related information

–

# –w

## Command line syntax

**–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 suppresses warnings above level 5:

```
arm16c –x –w5 lib.a obj1.obj
```

## Related information


–

# –x

## Command line syntax

**–x** [**–o**] [**–v**]

## Description

Extract an existing module from the library.

**–o** Give the extracted object module the same date as the last–modified date that was recorded in the library.

Without this suboption it receives the last–modified date of the moment it is extracted.

**–v** Verbose: the archiver shows which files are extracted.

## Example

To extract the file `obj.obj` from the library `lib.a`:

```
arm16c –x lib.a obj1.obj
```

If you do not specify an object module, all object modules are extracted:

```
arm16c –x lib.a
```

## Related information

–

## 4.7 FLASH UTILITY OPTIONS

The flash utility **flashm16c** is a tool to load an ELF, IEEE–695, Intel Hex or Motorola S–record file in a flash device. Normally, you would invoke it through EDE but you can also use it from the command line.

The invocation syntax is:

**flashm16c** [*option*]... [*file*]...

# -actions

## Command line syntax

**–actions=***flag...*

## Description

With this option you can specify the actions you want to perform with the flash tool. You can specify one or more of the following flags:

**B** Black check. Use this to check if the flash device is properly erased.

**F** Full erase. Use this to erase the entire flash memory.

**P** Program blocks. Use this to program the flash device with the specified file.

**V** Verify programmed block. Use this to compare an absolute file with the content of the FLASH.

## Example

```
flashm16c –actions=FP –baudrate=38400 –com2
          –id=00.00.00.00.00.00.00.00 –nodialog demo.s
```

This erases the flash device and flashes the file `demo.s` at a baud rate of 38400 in a device connected at serial port COM2.

## Related information

–

# -backup

## Command line syntax

**–backup** *file*

## Description

With this option you can save the original contents of the FLASH device before overwriting it.

## Example

```
flashm16c –backup backup.hex –actions=FP –nodialog
     –baudrate=38400 –com2 –id=00.00.00.00.00.00.00.00
     demo.s
```

The flash utility saves the contents of the FLASH device in file `backup.hex`.

## Related information

Flash utility option **–backup_range**

# -backup_range

### Command line syntax

**–backup_range=***start*,*end*

### Description

With this option you can specify the start and end address for the backup. The default bakcup range is 0xC0000 – 0xFFFFF. Use this option in combination with **–backup**.

### Example

```
flashm16c –actions=FP –nodialog –baudrate=38400 –com2
     –backup backup.hex –backup_range=0xc0000,0xdffff
     –id=00.00.00.00.00.00.00 demo.s
```

The flash utility saves address range 0xC0000 – 0xDFFFF in file `backup.hex`.

### Related information

Flash utility option **–backup**

# -baudrate

## Command line syntax

**–baudrate=***baudrate*

## Description

With this option you can specify the baud rate for serial port communication. Allowed values are: 9600, 19200, 38400 and 57600. The default is 9600. This option only works in combination with a **–com** option.

## Example

```
flashm16c –actions=FP –baudrate=38400 –com2
          –id=00.00.00.00.00.00.00 –nodialog demo.s
```

This erases the flash device and flashes the file `demo.s` at a baud rate of 38400 in a device connected at serial port COM2.

## Related information

Flash utility option **–com**

# -com

## Command line syntax

**–com**{**1** | **2** | **3** | **4**}

## Description

With this option you can specify the serial communication port COM1, COM2, COM3 or COM4. The default is COM1.

## Example

```
flashm16c –actions=FP –baudrate=38400 –com2
          –id=00.00.00.00.00.00.00 –nodialog demo.s
```

This erases the flash device and flashes the file `demo.s` at a baud rate of 38400 in a device connected at serial port COM2.

## Related information

Flash utility option **–baudrate**

# -dir

## Command line syntax

**–dir** *path*

## Description

Normally you must call the flash utility **flashm16c** from the directory where your project files are stored.

With the option **–dir** you can call the flash utility from within another directory. The *path* is the path to the directory where your absolute object file is stored and can be absolute or relative to your current directory.

## Example

Suppose your project files are stored in the directory `..\myfiles`. You can call the flash utility as follows:

```
flashm16c –dir ..\myfiles –actions=FP –baudrate=38400
     –com2 –id=00.00.00.00.00.00.00 –nodialog demo.s
```

## Related information


–

# -err

## Command line syntax

**–err** *file*

## Description

With this option the flash utility redirects error messages and verbose messages to a specified file.

## Example

```
flashm16c –err error.txt
```

The flash utility writes messages to the file `error.txt`.

## Related information

Flash utility option **–level**

# –f

## Command line syntax

**–f** *file*

## Description

Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the flash utility **flashm16c**.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

Option files can also be generated on the fly, for example by the make utility. You can specify the option **–f** multiple times.

### *Format of an option file*

- Multiple arguments on one line in the option file are allowed.
- To include whitespace in an argument, surround the argument with single or double quotes.
- If you want to use single quotes as part of the argument, surround the argument by double quotes and 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:

```
-actions=FP
-baudrate=38400
-com2
-id=00.00.00.00.00.00.00.00
-nodialog
demo.s
```

Specify the option file to the librarian:

```
flashm16c -f myoptions
```

This is equivalent to the following command line:

```
flashm16c -actions=FP -baudrate=38400 -com2
          -id=00.00.00.00.00.00.00.00 -nodialog demo.s
```

### Related information

–

# -h

## Command line syntax

**–h**

## Description

Displays an overview of all command line options.

## Example

The following invocations display a list of the available command line options:

```
flashm16c –h
```

## Related information


–

# -id

## Command line syntax

**–id=***id*

## Description

With this option you can specify the seven bytes of the Flash ID Code (*ID1.ID2.ID3.ID4.ID5.ID6.ID7*). The flash utility sends this identification code to the flash device. If the code does not match the Flash ID Code on the flash device you cannot access the flash device.

## Example

```
flashm16c –actions=FP –baudrate=38400 –com2
          –id=00.00.00.00.00.00.00 –nodialog demo.s
```

This erases the flash device and flashes the file `demo.s` at a baud rate of 38400 in a device connected at serial port COM2.

## Related information

Flash utility option **–noidretry**

# -level

### Command line syntax

**–level={0 | 1 | 2 | 3}**

### Description

With this you can specify the amount of detail in which you want to see messages. Normally, you would only use this to detect the cause when errors occur. The default level is 0 (no logging). Level 3 logging shows the most details.

### Example

To flash and log all messages when errors occur:

```
flashm16c –actions=FP –level=3 –nodialog
          –id=00.00.00.00.00.00.00 demo.s
```

### Related information

Flash utility option **–err**

# -M16C10 / -R8C10

## Command line syntax

**–M16C10**

**–R8C10**

## Description

By default the flash utility assumes you are using a processor in the family M16C20 – M16C60. With these options you select the M16C10 or the R8C10 respectively.

## Example

```
flashm16c –actions=FP –M16C10 –USB –nodialog
          –id=00.00.00.00.00.00.00 demo.s
```

This erases the flash device and flashes the file **demo.s** using USB communication. The target is an M16C10.

## Related information

–

# -nodialog

### Command line syntax

**–nodialog**

### Description

With this option you prevent the graphical user interface of the flash utility to come up. EDE uses this option by default. Without this opton you have to click the **Start** button in the flash tab to start flashing.

### Example

```
flashm16c –actions=FP –baudrate=38400 –com2
          –id=00.00.00.00.00.00.00.00 –nodialog demo.s
```

This erases the flash device and flashes the file `demo.s` at a baud rate of 38400 in a device connected at serial port COM2.

### Related information


–

# -noidretry

## Command line syntax

**–noidretry**

## Description

If the Flash ID Code consists of all 00 or FF, the flash utility tries the opposite if the Flash ID Code fails. With this option the flash utility does not try with the opposite.

## Example

```
flashm16c –actions=FP –nodialog
          –id=00.00.00.00.00.00.00 –noidretry demo.s
```

This erases the flash device and flashes the file `demo.s`. If the flash ID (all 00) does not match, the flash utility does not retry with the opposite (all FF).

## Related information

Flash utility option **–id**

# -set_USB_target

### Command line syntax

**–set_USB_target=***target*

### Description

With this option you specify the target board and download the file *target*.s to the USB monitor board. This hex file contains the firmware for the board. See the `usb` directory relative to the installation directory for the available targets.

### Example

```
flashm16c –actions=FP –USB –set_USB_target=r5f21114usb
          –id=00.00.00.00.00.00.00 –nodialog demo.s
```

This erases the flash device and flashes the file `demo.s` using USB communication.

### Related information


–

# -USB

## Command line syntax

**–USB**

## Description

With this option you specify to use an USB connection.

## Example

```
flashm16c –actions=FP –USB –id=00.00.00.00.00.00.00.00
          –nodialog demo.s
```

This erases the flash device and flashes the file `demo.s` using USB communication.

## Related information

Flash utility option **–com**

# -version

### Command line syntax

**–version**

### Description

Displays version information of the flash software that executes on the target board (instead of the flash utility itself).

### Example

```
flashm16c –version
```

### Related information

–

CHAPTER 5

# LIST FILE FORMATS

TASKING

# 5 CHAPTER

## 5.1 ASSEMBLER LIST FILE FORMAT

The assembler list file is an additional output file of the assembler that contains information about the generated code.

The list file consists of a page header and a source listing.

### *Page header*

The page header consists of four lines:

```
TASKING M16C Assembler vx.yrz Build nnn SN 00000000
This is the page header title                              Page 1

ADDR CODE      CYCLES  LINE SOURCE LINE
```

The first line contains information about the assembler name, version number and serial number. The second line contains a title specified by the TITLE (first page) assembler directive and a page number. The third line is empty. The fourth line contains the heading of the source listing.

### *Source listing*

The following is a sample part of a listing. An explanation of the different columns follows below.

```
ADDR CODE      CYCLES  LINE SOURCE LINE
                         .
                         .
0000 754Frrrr  4     4    14         push.w  _world
0004 A2rrrr    2     6    15         mov.w   #__2_ini, A0
0007 FDrrrr0r  9    15    16         jsr     _printf
000B 7DB2      1    16    17         add.b   #2, SP
000D F3        6    22    18         rts
                         .
                         .
0000                     33 _world:
0000                     34         ds  4
  |  RESERVED
0003
```

The meaning of the different columns is:

`ADDR` This column contains the memory address. The address is a hexadecimal number that represents the offset from the beginning of a relocatable section or the absolute address for an absolute section. The address only appears on lines that generate object code.

`CODE` This is the object code generated by the assembler for this source line, displayed in hexadecimal format. The displayed code need not be the same as the generated code that is entered in the object module. The code can also be relocatable code. In this case the letter 'r' is printed for the relocatable code part in the listing. For lines that allocate space, the code field contains the text "RESERVED". For lines that initialize a buffer, the code field lists one value followed by the word "REPEATS".

`CYCLES` The first number in this column is the number of instruction cycles needed to execute the instruction(s) as generated in the `CODE` field. The second number is the accumulated cycle count of this section.

`LINE` This column contains the line number. This is a decimal number indicating each input line, starting from 1 and incrementing with each source line.

`SOURCE LINE` This column contains the source text. This is a copy of the source line from the assembly source file.

For the `SET` and `EQU` directives the `ADDR` and `CODE` columns do not apply. The symbol value is listed instead.

***Related information***

See section 6.6, *Generating a List File*, in Chapter *Using the Assembler* of the *User's Guide* for more information on how to generate a list file and specify the amount of list file information.

## 5.2 LINKER MAP FILE FORMAT

The linker map file is an additional output file of the linker that shows how the link phase has mapped the sections and symbols from the various object files (`.obj`) to output sections. The locate part shows the absolute position of each section. External symbols are listed per space with their absolute address, both sorted on symbol and sorted on address.

With the linker option **–m** (map file formatting) you can specify which parts of the map file you want to see.

### *Example (part of) linker map file*

```
M16C linker - mapfile (task1)
Options: -o hello.elf -M -m2 -mcfklmoqrsu --map-file
-------------------------------------------------------------------------------

************************************  File Part  **********************************

* Processed files:
==================

  File        | From archive | Symbol causing the extraction
   ------------------------------------------------------------
  cstart.obj | libc.a        | __START
  hello.obj  |               |
  printf.obj | libc.a        | _printf



**************************************  Link Part  ********************************

* Section translation:
======================

  [in] File    | [in] Section      | [in] Size  | [out] Offset | [out] Section
   ------------------------------------------------------------------------------
  cstart.obj  | .cstart           | 0x0000001f | 0x00000000   | .cstart
   ------------------------------------------------------------------------------
  hello.obj   | hello_CO          | 0x0000000e | 0x00000000   | hello_CO
   ------------------------------------------------------------------------------
  hello.obj   | hello_CO          | 0x00000014 | 0x00000000   | hello_CO
   ------------------------------------------------------------------------------
  printf.obj  | printf_CO         | 0x0000001e | 0x00000000   | printf_CO
```

```
*******************************  Module Local Symbols Part  *************************

* Local symbol translation (sorted on symbol):
==============================================

  + File "_doprint_int.obj"
  + Scope "/usr/src/m32c/dvl/linux/cm16c/lib/m16c/libcs.a"

  Symbol             | Address    | Space
   ------------------------------------------------
  _doprint_int.src   | 0x00000000 |
   ------------------------------------------------
  _10                | 0x000c0111 | M16C:M16C:far
  _100               | 0x000c055c |
  _101               | 0x000c0597 |

*********************************  Cross Reference Part  ****************************

* Defined symbols:
==================

  Definition file | Definition section    | Symbol     | Referenced in
   ------------------------------------------------------------------------------------
  cstart.obj      | .cstart               | __Exit     | exit.obj
  cstart.obj      | .cstart               | __START    | hello.obj
  hello.obj       | hello_CO              | _main      | cstart.obj, exit.obj, ....


* Undefined symbols:
====================

  Symbol          | Referenced in
   ------------------------------------------------------------------------------------
  __init          | cstart.obj
  __lc_es         | cstart.obj
  __vecttab       | cstart.obj
```

```
************************************  Locate Part  *********************************

* Task entry address:
====================

  symbol : __START

* Section translation:
=====================

  + Space M16C:M16C:far

  Chip     | Group | Section             | Size (MAU) | Space addr | Chip addr
   ------------------------------------------------------------------------------
  ram_mem | sfr   | sfr                 | 0x00000400 | 0x00000000 | 0x00000000
  rom_mem |       | [_iob_INI_DA]       | 0x0000006e | 0x000c0000 | 0x00000000
          |       | [fss_init_INI_DA]   | 0x0000000c | 0x000c006e | 0x0000006e
          |       | [hello_INI_DA]      | 0x00000014 | 0x000c007a | 0x0000007a
          |       | table               | 0x00000032 | 0x000c008e | 0x0000008e
          |       | hello_CO            | 0x0000000e | 0x000c00c0 | 0x000000c0
          |       | .cstart             | 0x0000001f | 0x000c00ce | 0x000000ce
          |       | exit_CO             | 0x00000017 | 0x000c00ed | 0x000000ed
          |       | printf_int_CO       | 0x00000001 | 0x000c0104 | 0x00000104

* Symbol translation (sorted on symbol):
========================================

  Symbol             | Address     | Space
   ------------------------------------------------
  __dcti             | 0x00000000 |
  __vecttab          | 0x00000000 |
   ------------------------------------------------
  __Exit             | 0x000c00eb | M16C:M16C:far
  __START            | 0x000c00ce |

* Symbol translation (sorted on address):
=========================================

  Address     | Symbol             | Space
   ------------------------------------------------
  0x00000000 | __vecttab          |
  0x00000000 | __dcti             |
   ------------------------------------------------
  0x00000000 | __lc_gb_sfr        | M16C:M16C:far
  0x00000400 | __lc_ge_sfr        |
```

```
************************************  Memory Part  *********************************

  Name                  | Total      | Used        %  | Free        % |  > free gap  %
  ----------------------------------------------------------------------------------
  M16C:M16C:bit         | 0x00010000 | 0x0000e000  88 | 0x00002000  12|  0x00002000  12
  M16C:M16C:bita        | 0x00002000 | 0x00001c00  88 | 0x00000400  12|  0x00000400  12
  M16C:M16C:far         | 0x00100000 | 0x00005128   2 | 0x000faed8  98|  0x000bb9c3  73
  M16C:M16C:near        | 0x00010000 | 0x0000423d  26 | 0x0000bdc3  74|  0x0000b9c3  72

* Address range usage at memory level:
======================================

  Name                  | Total      | Used        %  | Free        % |  > free gap  %
  ----------------------------------------------------------------------------------
  ram_mem               | 0x00040000 | 0x0000423d   7 | 0x0003bdc3  93|  0x0003b9c3  93
  rom_mem               | 0x00040000 | 0x00000eeb   2 | 0x0003f115  98|  0x0003f115  98

*******************************  Linker Script File Part  ***************************


*********************************  Locate Rule Part  ********************************

  Address space | Type             | Properties                  | Sections
  ----------------------------------------------------------------------------------
  M16C:M16C:far  | unrestricted |                               | table   hello_CO ...
  M16C:M16C:far  | clustered    |                               | [_iob_INI_DA] + ...
  M16C:M16C:far  | absolute     | 0x000ffffc                    | .reset
  M16C:M16C:far  | absolute     | 0x00000000                    | sfr
```

The meaning of the different parts is:

### *File Part*

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 Part: Section translation*

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 from the input object file. |
| `[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. |

### *Module Local Symbols Part*

This part of the map file shows a table for each local scope within an object file. Each table has three columns, 1 the symbolname, 2 the address of the symbol and 3 the space where the symbol resides in. The table is sorted on symbolname within each space.

By default this part is not shown in the map file. You have to turn this part on manually with linker option **–m2** or **–mq** (module local symbols).

### *Cross Reference Part*

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.

By default this part is not shown in the map file. You have to turn this part on manually with linker option **–mr** (cross references info).

### *Locate Part: Section translation*

This part of the map file shows the absolute position of each section in the absolute object file. It is organized per address space, memory chip and group and sorted on space address.

| | |
|---|---|
| `+ Space` | The names of the address spaces as defined in the linker script file (`*.lsl`). The names are constructed of the `derivative` name followed by a colon ':', the `core` name, another colon ':' and the `space` name. For example: `M16C:M16C:far` |
| `Chip` | The names of the memory chips as defined in the linker script file (`*.lsl`) in the `memory` definitions. |
| `Group` | Sections can be ordered in groups. These are the names of the groups as defined in the linker script file (`*.lsl`) with the keyword `group` in the `section_layout` definition. The name that is displayed is the name of the deepest nested group. |
| `Section` | The name of the section. Names within square brackets **[ ]** will be copied during initialization from ROM to the corresponding section name in RAM. |
| `Size (MAU)` | The size of the section in minimum addressable units. |

`Space addr` The absolute address of the section in the address space.

`Chip addr` The absolute offset of the section from the start of a memory chip.

### *Locate Part: Symbol translation*

This part of the map file lists all external symbols per address space name, both sorted on address and sorted on symbol name.

`Symbol` The name of the symbol.

`Address` The absolute address of the symbol in the address space.

`Space` The names of the address spaces as defined in the linker script file (`*.lsl`). The names are constructed of the `derivative` name followed by a colon ':', the `core` name, another colon ':' and the `space` name. For example: `M16C:M16C:far`

### *Memory Part*

This part of the map file shows the memory usage in totals and percentages for spaces and chips. The largest free block of memory per space and per chip is also shown.

By default this part is not shown in the map file. You have to turn this part on manually with linker option **–mm** (memory usage info).

### *Linker Script File Part*

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 **–ms** (processor and memory info). You can print this information to a separate file with linker option **––lsl–dump**.

### *Locate Rule Part*

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. For example: `M16C:M16C:far`

`Type` The rule type:

- `ordered/contiguous/clustered` Specifies how sections are grouped.
- `absolute address` The section must be located at the address shown in the Properties column
- `address range` The section must be located in the union of the address ranges shown in the Properties column; end addreses are not included in the range.
- `address range size` The sections must be located in some address range with size not larger than shown in the Properties column; the second number in that field is the alignment requirement for the address range.

`Properties` The contents depends on the Type column.

`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.

By default this part is not shown in the map file. You have to turn this part on manually with linker option **–mu** (locate rules).

### *Related information*

Section 7.8, *Generating a Map File*, in Chapter *Using the Linker* of the *User's Guide*.

Linker option **–M** (Generate map file)

CHAPTER 6

# OBJECT FILE FORMATS

TASKING

# 9 CHAPTER

## 6.1 ELF/DWARF OBJECT FORMAT

The M16C toolchain by default produces objects in the ELF/DWARF 2 (`.elf`) format.

For a complete description of the ELF and DWARF formats, please refer to the *Tools Interface Standards* on Intel's website for developers: http://developer.intel.com/vtune/tis.htm

## 6.2 MOTOROLA S-RECORD FORMAT

With the linker option **–o***filename***:SREC** option the linker produces output in Motorola S–record format with three types of S–records: S0, S2 and S8. With the options **–o***filename***:SREC:2** or **–o***filename***:SREC:4** option you can force other types of S–records. They have the following layout:

### *S0 – record*

'S' '0' <*length_byte*> <2 bytes 0> <*comment*> <*checksum_byte*>

A linker generated S–record file starts with a S0 record with the following contents:

length_byte : 0x9
comment : lkm16c
checksum : 0xE8

```
         l k m 1 6 c
S00900006C6B6D313663E8
```

The S0 record is a comment record and does not contain relevant information for program execution.

The length_byte represents the number of bytes in the record, not including the record type and length byte.

The checksum is calculated by first adding the binary representation of the bytes following the record type (starting with the length_byte) to just before the checksum. Then the one's complement is calculated of this sum. The least significant byte of the result is the checksum. The sum of all bytes following the record type is 0xFF.

### *S1 – record*

With the linker option **–o***filename***:SREC:2**, the actual program code and data is supplied with S1 records, with the following layout:

'S' '1' <*length_byte*> <*address*> <*code bytes*> <*checksum_byte*>

This record is used for 2–byte addresses.

Example:

```
S1130250F03EF04DF0ACE8A408A2A013EDFCDB00E6
  | |   |                                  |_ checksum
  | |   |_ code
  | |_ address
  |_ length
```

The length of the output buffer for generating S1 records is 32 code bytes.

The checksum calculation of S1 records is identical to S0.

### *S2 – record*

With the linker option **–o***filename***:SREC:3**, which is the default, the actual program code and data is supplied with S2 records, with the following layout:

'S' '2' <*length_byte*> <*address*> <*code bytes*> <*checksum_byte*>

By default, the linker generates 3–byte addresses.

Example:

```
S213FF002000232222754E00754F04AF4FAE4E22BF
  | |     |                                |_ checksum
  | |     |_ code
  | |_ address
  |_ length
```

The length of the output buffer for generating S2 records is 32 code bytes.

The checksum calculation of S2 records is identical to S0.

### *S3 – record*

With the linker option **–o***filename***:SREC:4**, the actual program code and data is supplied with S3 records, with the following layout:

'S' '3' <*length_byte*> <*address*> <*code bytes*> <*checksum_byte*>

This record is used for 4–byte addresses.

Example:

```
S3070000FFFE6E6825
  | |        |   |_ checksum
  | |        |_ code
  | |_ address
  |_ length
```

The length of the output buffer for generating S3 records is 32 code bytes.

The checksum calculation of S3 records is identical to S0.

### *S7 – record*

With the linker option **–o***filename***:SREC:4**, at the end of an S–record file, the linker generates an S7 record, which contains the program start address. S7 is the corresponding termination record for S3 records.

Layout:

'S' '7' <*length_byte*> <*address*> <*checksum_byte*>

Example:

```
S70500006E6824
  | |        |_checksum
  | |_ address
  |_ length
```

The checksum calculation of S7 records is identical to S0.

### *S8 – record*

With the linker option **–o***filename***:SREC:3**, which is the default, at the end of an S–record file, the linker generates an S8 record, which contains the program start address.

Layout:

'S' '8' <*length_byte*> <*address*> <*checksum_byte*>

Example:

```
S804FF0003F9
  | |      |_checksum
  | |_ address
  |_ length
```

The checksum calculation of S8 records is identical to S0.

### *S9 – record*

With the linker option **–o***filename***:SREC:4**, at the end of an S–record file, the linker generates an S9 record, which contains the program start address. S9 is the corresponding termination record for S1 records.

Layout:

'S' '9' <*length_byte*> <*address*> <*checksum_byte*>

Example:

```
S9030210EA
  | |    |_checksum
  | |_ address
  |_ length
```

The checksum calculation of S9 records is identical to S0.

## 6.3 INTEL HEX RECORD FORMAT

Intel Hex records describe the hexadecimal object file format for 8–bit, 16–bit and 32–bit microprocessors. The hexadecimal object file is an ASCII representation of an absolute binary object file. There are six different types of records:

- Data Record (8–, 16, or 32–bit formats)
- End of File Record (8–, 16, or 32–bit formats)
- Extended Segment Address Record (16, or 32–bit formats)
- Start Segment Address Record (16, or 32–bit formats)
- Extended Linear Address Record (32–bit format only)
- Start Linear Address Record (32–bit format only)

For the M16C the linker generates records in the 32–bit format (4–byte addresses with linker option **–o***filename***:IHEX**).

### *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 that controls the length of the output buffer for generating Data records. The default buffer length is 32 bytes.

The *offset* is the 16–bit starting load offset. Together with the address specified in the Extended Address Record it specifies an absolute address in memory where the data is to be located when loaded by a tool.

Example:

```
:0F00200000232222754E00754F04AF4FAE4E22C3
 | |    | |                                |_ checksum
 | |    | |_ data
 | |    |_ type
 | |_ offset
 |_ length
```

### *Start Linear Address Record*

The Start Linear Address Record contains the 32–bit program execution start address.

Layout:

| : | 04 | 0000 | 05 | *address* | *checksum* |
|---|---|---|---|---|---|

Example:

```
:040000050000CE2405
 | |    | |       |_ 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
```

CHAPTER 7

# LINKER SCRIPT LANGUAGE

TASKING

# 1 CHAPTER

## 7.1 INTRODUCTION

To make full use of the linker, you can write a script with information about the architecture of the target processor and locating information. The language for the script is called the *Linker Script Language* (LSL). This chapter first describes the structure of an LSL file. The next section contains a summary of the LSL syntax. Finally, in the remaining sections, the semantics of the Linker Script Language is explained.

The TASKING linker is a target independent linker/locator that can simultaneously link and locate all programs for all cores available on a target board. The target board may be of arbitrary complexity. A simple target board may contain one standard processor with some external memory that executes one task. A complex target board may contain multiple standard processors and DSPs combined with configurable IP–cores loaded in an FPGA. Each core may execute a different program, and external memory may be shared by multiple cores.

LSL serves two purposes. First it enables you to specify the characteristics (that are of interest to the linker) of your specific target board and of the cores installed on the board. Second it enables you to specify how sections should be located in memory.

## 7.2 STRUCTURE OF A LINKER SCRIPT FILE

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

***The architecture definition (required)***

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

This specification is normally written by Altium. 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 7.5, *Semantics of the Architecture Definition* for detailed descriptions of LSL in the architecture definition.

### *The derivative definition (required)*

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

See section 7.6, *Semantics of the Derivative Definition* for a detailed description of LSL in the derivative definition.

### *The processor definition*

The *processor definition* describes an instance of a derivative. Typically the processor definition instantiates one derivative only (single–core processor). A processor that contains multiple cores having the same (homogeneous) or different (heterogeneous) architecture can also be described by instantiating multiple derivatives of the same or different types in separate processor definitions.

See section 7.7, *Semantics of the Board Specification* for a detailed description of LSL in the processor definition.

### *The memory and bus definitions (optional)*

Memory and bus definition are used within the context of a derivative definition to specify internal memory and on–chip buses. In the context of a board specification the memory and bus definitions are used to define external (off–chip) memory and buses. Given the above definitions the linker can convert a logical address into an offset into an on–chip or off–chip memory device.

See section 7.7.3, *Defining External Memory and Buses*, for more information on how to specify the external physical memory layout. *Internal* memory for a processor should be defined in the derivative definition for that processor.

### *The board specification*

The processor definition and memory and bus definitions together form a *board specification*. LSL provides language constructs to easily describe single–core and heterogeneous or homogeneous multi–core systems. The board specification describes all characteristics of your target board's system buses, memory devices, I/O sub–systems, and cores that are of interest to the linker. Based on the information provided in the board specification the linker can for each core:

- convert a logical address to a physical addresses (offsets 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 7.8, *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*

The skeleton of a linker script file now looks as follows:

```
architecture architecture_name
{
    architecture definition
}

derivative derivative_name
{
    derivative definition
}
```

```
processor processor_name
{
    processor definition
}

memory definitions and/or bus definitions

section_layout space_name
{
    section placement statements
}
```

## 7.3 SYNTAX OF THE LINKER SCRIPT LANGUAGE

### 7.3.1 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`.

For example:

```
#include "arch.lsl"
```

Preprocess and include the file `arch.lsl` at this point in the LSL file.

### 7.3.2 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 '`//`'.

### 7.3.3 IDENTIFIERS

```
arch_name          ::= IDENTIFIER
bus_name           ::= IDENTIFIER
core_name          ::= IDENTIFIER
derivative_name    ::= IDENTIFIER
file_name          ::= DQSTRING
group_name         ::= IDENTIFIER
mem_name           ::= IDENTIFIER
proc_name          ::= IDENTIFIER
section_name       ::= DQSTRING
space_name         ::= IDENTIFIER
stack_name         ::= section_name
symbol_name        ::= DQSTRING
```

### 7.3.4 EXPRESSIONS

The expressions and operators in this section work the same as in ANSI 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
```

### 7.3.5 BUILT-IN FUNCTIONS

```
function_call   ::= absolute ( expr )
                  | addressof ( addr_id )
                  | exists ( section_name )
                  | max ( expr , expr )
                  | min ( expr , expr )
                  | sizeof ( size_id )

addr_id         ::= sect : section_name
                  | group : group_name
```

```
size_id          ::= group : group_name
                   | mem : mem_name
                   | sect : section_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.

## 7.3.6 LSL DEFINITIONS IN THE LINKER SCRIPT FILE

```
description       ::= <definition>>=1

definition         ::= architecture_definition
                     | derivative_definition
                     | board_spec
                     | section_definition
```

- At least one *architecture_definition* must be present in the LSL file.

### 7.3.7 MEMORY AND BUS DEFINITIONS

```
mem_def          ::= memory mem_name { <mem_descr ;>>=0 }
```

- A *mem_def* defines a *memory* with the *mem_name* as a unique name.

```
mem_descr        ::= type = <reserved>0|1 mem_type
                   | mau = expr
                   | size = expr
                   | speed = number
                   | 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 (default value is 1).
- A *mem_def* contains at least one *mapping*.

```
mem_type         ::= rom          // attrs = rx
                   | ram          // attrs = rw
                   | nvram        // attrs = rwx
```

```
bus_def          ::= bus bus_name { <bus_descr ;>>=0 }
```

- A *bus_def* statement defines a *bus* with the given *bus_name* as a unique name within a core architecture.

```
bus_descr        ::= mau = expr
                   | width = expr  // bus width, nr
                   |               // of data bits
                   | mapping       // legal destination
                                   // 'bus' only
```

- The **mau** and **width** statements appear exactly once in a *bus_descr*. The default value for **width** is the **mau** size.
- The bus width must be an integer times the bus MAU size.
- The MAU size must be non–zero.
- A bus can only have a *mapping* on a destination *bus* (through **dest = bus:** ).

```
mapping          ::= map ( map_descr <, map_descr>>=0 )
```

```
map_descr          ::= dest = destination
                     | dest_dbits = range
                     | dest_offset = expr
                     | size = expr
                     | src_dbits = range
                     | src_offset = expr
```

- A *mapping* requires at least the **size** and **dest** statements.
- Each *map_descr* can occur only once.
- You can define multiple mappings from a single source.
- Overlap between source ranges or destination ranges is not allowed.
- If the **src_dbits** or **dest_dbits** statement is not present, its value defaults to the **width** value if the source/destination is a bus, and to the **mau** size otherwise.

```
destination        ::= space : space_name
                     | bus : <proc_name |
                              core_name :>0|1 bus_name
```

- A *space_name* refers to a defined address space.
- A *proc_name* refers to a defined processor.
- A *core_name* refers to a defined core.
- A *bus_name* refers to a defined bus.
- The following mappings are allowed (source to destination)
  - space => space
  - space => bus
  - bus => bus
  - memory => bus

```
range              ::= number .. number
```

### 7.3.8 ARCHITECTURE DEFINITION

```
architecture_definition
                   ::= architecture arch_name
                       <( parameter_list )>0|1
                       <extends arch_name
                                <( argument_list )>0|1 >0|1
                       { arch_spec>=0 }
```

- An *architecture_definition* defines a core *architecture* with the given *arch_name* as a unique name.

- At least one *space_def* and at least one *bus_def* have to be present in an *architecture_definition*.
- An *architecture_definition* that uses the **extends** construct defines an architecture that *inherits* all elements of the architecture defined by the second *arch_name*. The *parent architecture* must be defined in the LSL file as well.

```
parameter_list    ::= parameter <, parameter>^>=0

parameter         ::= IDENTIFIER <= expr>^0|1

argument_list     ::= expr <, expr>^>=0

arch_spec         ::= bus_def
                    | space_def
                    | endianness_def

space_def         ::= space space_name { <space_descr;>^>=0 }
```

- A *space_def* defines an address space with the given *space_name* as a unique name within an architecture.

```
space_descr       ::= space_property ;
                    | section_definition  //no space ref

space_property    ::= id = number // as used in object
                    | mau = expr
                    | align = expr
                    | page_size = expr
                    | stack_def
                    | heap_def
                    | copy_table_def
                    | start_address
                    | mapping
```

- A *space_def* contains exactly one **id** and one **mau** statement.
- A *space_def* contains at most one **align** statement.
- A *space_def* contains at most one **page_size** statement.
- A *space_def* contains at least one mapping.

```
stack_def         ::= stack stack_name ( stack_heap_descr
                          <, stack_heap_descr >^>=0 )
```

- A *stack_def* defines a stack with the *stack_name* as a unique name.

```
heap_def          ::= heap heap_name ( stack_heap_descr
                          <, stack_heap_descr >^>=0 )
```

- A *heap_def* defines a heap with the *heap_name* as a unique name.

```
copy_table_def    ::= copytable ( copy_table_descr
                                  <, copy_table_descr>>=0 )
```

- A *space_def* contains at most one **copytable** statement.
- If the architecture definition contains more than one address space, exactly one copy table must be defined in one of the spaces. If the the architecture definition contains only one address space, a copy table definition is optional (it will be generated in the space).

```
stack_heap_descr  ::= min_size = expr
                    | grows = direction
                    | align = expr
                    | fixed
```

- The **min_size** statement must be present.
- You can specify at most one **align** statement and one **grows** statement.

```
direction         ::= low_to_high
                    | high_to_low
```

- If you do not specify the **grows** statement, the stack and grow **low-to-high**.

```
copy_table_descr  ::= align = expr
                    | copy_unit = expr
                    | dest <space_name>0|1 = space_name
```

- 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.

```
endianness_def    ::= endianness { <endianness_type;>>=1 }
```

```
endianness_type   ::= big
                    | little
```

### 7.3.9 DERIVATIVE DEFINITION

```
derivative_definition
                  ::= derivative derivative_name
                      <( parameter_list )>0|1
                      <extends derivative_name
                              <( argument_list )>0|1 >0|1
                      { <derivative_spec>>=0 }
```

- A *derivative_definition* defines a derivative with the given *derivative_name* as a unique name.
- At least one *core_def* must be present in a *derivative_definition*.

```
derivative_spec   ::= core_def
                    | bus_def
                    | mem_def
                    | section_definition // no processor
                                         // name

core_def          ::= core core_name { <core_descr ;>>=0 }
```

- A *core_def* defines a *core* with the given *core_name* as a unique name.

```
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*.

### 7.3.10 PROCESSOR DEFINITION AND BOARD SPECIFICATION

```
board_spec        ::= proc_def
                    | bus_def
                    | mem_def
```

```
proc_def          ::= processor proc_name
                      { proc_descr ; }

proc_descr        ::= derivative = derivative_name
                      <( argument_list )>0|1
```

- A *proc_def* defines a *processor* with the *proc_name* as a unique name.
- If you do not explicitly define a processor for a derivative in an LSL file, the linker defines a processor with the same name as that derivative.
- A *derivative_name* refers to a defined derivative.
- A *proc_def* contains exactly one **derivative** statement.



## 7.3.11 SECTION PLACEMENT DEFINITION

```
section_definition ::= section_layout <space_ref>0|1
                       <( locate_direction )>0|1
                       { <section_statement>>=0 }
```

- A section definition inside a space definition does not have a *space_ref*.
- All global section definitions have a *space_ref*.

```
space_ref         ::= <proc_name>0|1 : <core_name>0|1
                      : space_name
```

- If more than one processor is present, the *proc_name* must be given for a global section layout.
- If the section layout refers to a processor that has more than one core, the *core_name* must be given in the *space_ref*.
- A *proc_name* refers to a defined processor.
- A *core_name* refers to a defined core.
- A *space_name* refers to a defined address space.

```
locate_direction  ::= direction = direction

direction         ::= low_to_high
                    | high_to_low
```

- A section layout contains at most one **direction** statement.
- If you do not specify the **direction** statement, the locate direction of the section layout is **low–to–high**.

```
section_statement
                  ::= simple_section_statement ;
                    | aggregate_section_statement

simple_section_statement
                  ::= assignment
                    | select_section_statement
                    | special_section_statement

assignment        ::= symbol_name assign_op expr

assign_op         ::= =
                    | :=

select_section_statement
                  ::= select <section_name>0|1
                      <section_selections>0|1
```

- Either a *section_name* or at least one *section_selection* must be defined.

```
section_selections
                  ::= ( section_selection
                        <, section_selection>>=0 )

section_selection
                  ::= attributes = < <+|-> attribute>>0
```

- **+***attribute* means: select all sections that have this attribute.
- **-***attribute* means: select all sections that do not have this attribute.

```
special_section_statement
                  ::= heap stack_name <size_spec>0|1
                    | stack stack_name <size_spec>0|1
                    | copytable
                    | reserved <section_name>0|1
                               <reserved_specs>0|1
```

- Special sections cannot be selected in load–time groups.

```
size_spec         ::= ( size = expr )

reserved_specs    ::= ( reserved_spec
                        <, reserved_spec>>=0 )
```

```
reserved_spec      ::= attributes
                     | fill_spec
                     | size = expr
                     | alloc_allowed = absolute
```

- 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).

```
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
```

- No two groups for an address space can have the same *group_name*.

```
group_specs        ::= group_spec <, group_spec >>=0

group_spec         ::= group_alignment
                     | attributes
                     | group_load_address
                     | fill <= fill_values>0|1
                     | group_page
                     | group_run_address
                     | group_type
                     | allow_cross_references
```

- 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

group_load_address
                   ::= load_addr <= load_or_run_addr>0|1

fill_spec          ::= fill = fill_values

fill_values        ::= expr
                     | [ expr <, expr>>=0 ]
```

```
group_page          ::= page <= expr>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.

```
attribute           ::= r     // read-only sections
                      | w     // read/write sections
                      | x     // executable code sections
                      | i     // initialized sections
                      | s     // scratch sections
                      | b     // blanked (cleared) sections

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**.

```
memory_reference    ::= mem : <proc_name :>0|1
                              <core_name :>0|1 mem_name
```

- A *proc_name* refers to a defined processor.
- A *core_name* refers to a defined core.
- A *mem_name* refers to a defined memory.

```
if_statement        ::= if ( expr ) section_statement
                        <else section_statement>0|1
```

```
section_creation_statement
                    ::= section section_name
                                ( <section_spec>0|1 )
                        { <select_section_statement ;>>=0 }

section_spec        ::= attributes
                      | fill_spec
                      | size = expr
```

## 7.4 EXPRESSION EVALUATION

Only *constant* expressions are allowed, including sizes, but not addresses, of sections in object files.

All expressions are evaluated with 64–bit precision integer arithmetic. The result of an expression can be absolute or relocatable. A symbol you assign is created as an absolute symbol.

## 7.5 SEMANTICS OF THE ARCHITECTURE DEFINITION

***Keywords in the architecture definition***

```
architecture
   extends
endianness          big  little
bus
   mau
   width
   map
space
   id
   mau
   align
   page_size
   stack
      min_size
      grows         low_to_high  high_to_low
      align
      fixed
   heap
      min_size
      grows         low_to_high  high_to_low
      align
      fixed
   copytable
      align
      copy_unit
      dest
   start_address
      run_addr
      symbol
   map

   map
      dest          bus  space
      dest_dbits
      dest_offset
      size
      src_dbits
      src_offset
```

### 7.5.1 DEFINING AN ARCHITECTURE

With the keyword **architecture** you define an architecture and assign a unique name to it. The name is used to refer to it at other places in the LSL file:

```
architecture name
{
     definitions
}
```

If you are defining multiple core architectures that show great resemblance, you can define the common features in a parent core architecture and extend this with a child core architecture that contains specific features. The child inherits all features of the parent. With the keyword **extends** you create a child core architecture:

```
architecture name_child_arch extends name_parent_arch
{
     definitions
}
```

A core architecture can have any number of parameters. These are identifiers which get values assigned on instantiation or extension of the architecture. You can use them in any expression within the core architecture. Parameters can have default values, which are used when the core architecture is instantiated with less arguments than there are parameters defined for it. When you extend a core architecture you can pass arguments to the parent architecture. Arguments are expressions that set the value of the parameters of the sub–architecture.

```
architecture name_child_arch (parm1,parm2=1)
             extends name_parent_arch (arguments)
{
     definitions
}
```

### 7.5.2 DEFINING INTERNAL BUSES

With the **bus** keyword you define a bus (the combination of data and corresponding address bus). The bus name is used to identify a bus and does not conflict with other identifiers. Bus descriptions in an architecture definition or derivative definition define *internal* buses. Some internal buses are used to communicate with the components outside the core or processor. Such buses on a processor have physical pins reserved for the number of bits specified with the **width** statements.

- The **mau** field specifies the MAU size (Minimum Addressable Unit) of the data bus. This field is required.
- The **width** field specifies the width (number of address lines) of the data bus. The default value is the MAU size.
- The **map** keyword specifies how this bus maps onto another bus (if so). Mappings are described in section 7.5.4, *Mappings*.

```
bus bus_name
{
      mau = 8;
      width = 8;
      map ( map_description );
}
```

### 7.5.3 DEFINING ADDRESS SPACES

With the **space** keyword you define a logical address space. The space name is used to identify the address space and does not conflict with other identifiers.

- The **id** field defines how the addressing space is identified in object files. In general, each address space has a unique ID. The linker locates sections with a certain ID in the address space with the same ID. This field is required. In IEEE this ID is specified explicitly for sections and symbols, ELF sections map by default to the address space with ID 1. Sections with one of the special names defined in the ABI (Application Binary Interface) may map to different address spaces.
- The **mau** field specifies the MAU size (Minimum Addressable Unit) of the space. This field is required.
- The **align** value must be a power of two. The linker uses this value to compute the start addresses when sections are concatenated. An align value of *n* means that objects in the address space have to be aligned on *n* MAUs.

- The **page_size** field sets the page size in MAUs for the address space. It must be a power of 2. The default page size is 1. See also the **page** keyword in subsection *Locating a group* in section 7.8.2, *Creating and Locating Groups of Sections*.
- The **map** keyword specifies how this address space maps onto an internal bus or onto another address space. Mappings are described in section 7.5.4, *Mappings*.

### *Stacks and heaps*

- The **stack** keyword defines a stack in the address space and assigns a name to it. The architecture definition must contain at least one stack definition. Each stack of a core architecture must have a unique name. See also the **stack** keyword in section 7.8.3, *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.

  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 7.8.3, *Creating or Modifying Special Sections*.

See section 7.8, *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. If the architecture definition contains more than one address space, you must define exactly one copy table in one of the address spaces. If the architecture definition contains only one address space, the copy table definition is optional.

  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.

### *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);
      start_address ( run_addr = 0x0000,
                      symbol = "start_label" )
      map ( map_description );
}
```

### 7.5.4 MAPPINGS

You can use a mapping when you define a space, bus or memory. With the **map** field you specify how addresses from the source (space, bus or memory) are translated to addresses of a destination (space, bus). The following mappings are possible:

- space => space
- space => bus
- bus => bus
- memory => bus

With a mapping you specify a range of source addresses you want to map (specified by a source offset and a size), the destination to which you want to map them (a bus or another address space), and the offset address in the destination.

- The **dest** argument specifies the destination. This can be a **bus** or another address **space** (only for a space to space mapping). This argument is required.
- The **src_offset** argument specifies the offset of the source addresses. In combination with size, this specifies the range of address that are mapped. 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. Default the destination offset is 0x0000.

If you are mapping a bus to another bus, the number of data lines of each bus may differ. In this case you have to specify a range of source data lines you want to map (**src_dbits =** *begin*..*end*) and the range of destination data lines you want to map them to (**dest_dbits =** *first*..*last*).

- The **src_dbits** argument specifies a range of data lines of the source bus. Default all data lines are mapped.
- The **dest_dbits** argument specifies a range of data lines of the destination bus. Default, all data lines from the source bus are mapped on the data lines of the destination bus (starting with line 0).

### *From space to space*

If you map an address space to another address space (nesting), you can do this by mapping the subspace to the containing larger space. In this example a small space of 64k is mapped on a large space of 16M.

```
space small
{
   id = 2;
   mau = 4;
   map (src_offset = 0, dest_offset = 0,
        dest = space : large, size = 64k);
}
```

### *From space to bus*

All spaces that are not mapped to another space must map to a bus in the architecture:

```
space large
{
   id = 1;
   mau = 4;
   map (src_offset = 0, dest_offset = 0,
        dest = bus:bus_name, size = 16M );
}
```

### *From bus to bus*

The next example maps an external bus called **e_bus** to an internal bus called **i_bus**. This internal bus resides on a core called **mycore**. The source bus has 16 data lines whereas the destination bus has only 8 data lines. Therefore, the keywords **src_dbits** and **dest_dbits** specify which source data lines are mapped on which destination data lines.

```
architecture mycore
{
    bus i_bus
    {
       mau = 4;
    }

    space i_space
    {
       map (dest=bus:i_bus, size=256);
    }
}

bus e_bus
{
   mau = 16;
   width = 16;
   map (dest = bus:mycore:i_bus,
        src_dbits = 0..7, dest_dbits = 0..7 )
}
```

It is not possible to map an internal bus to an external bus.

## 7.6 SEMANTICS OF THE DERIVATIVE DEFINITION

***Keywords in the derivative definition***

```
derivative
   extends
core
   architecture
bus
   mau
   width
   map
memory
   type              rom  ram  nvram
   mau
   size
   speed
   map

   map
      dest           bus  space
      dest_dbits
      dest_offset
      size
      src_dbits
      src_offset
```

### 7.6.1 DEFINING A DERIVATIVE

With the keyword **derivative** you define a derivative and assign a unique name to it. The name is used to refer to it at other places in the LSL file:

```
derivative name
{
     definitions
}
```

If you are defining multiple derivatives that show great resemblance, you can define the common features in a parent derivative and extend this with a child derivative that contains specific features. The child inherits all features of the parent (cores and memories). With the keyword **extends** you create a child derivative:

```
derivative name_child_deriv extends name_parent_deriv
{
     definitions
}
```

As with a core architecture, a derivative can have any number of parameters. These are identifiers which get values assigned on instantiation or extension of the derivative. You can use them in any expression within the derivative definition.

```
derivative name_child_deriv (parm1,parm2=1)
           extends name_parent_derivh (arguments)
{
     definitions
}
```

### 7.6.2 INSTANTIATING CORE ARCHITECTURES

With the keyword **core** you instantiate a core architecture in a derivative.

- With the keyword **architecture** you tell the linker that the given core has a certain architecture. The architecture name refers to an existing architecture definition in the same LSL file.

  For example, if you have two cores (called `mycore_1` and `mycore_2`) that have the same architecture (called `mycorearch`), you must instantiate both cores as follows:

```
core mycore_1
{
     architecture = mycorearch;
}

core mycore_2
{
     architecture = mycorearch;
}
```

If the architecture definition has parameters you must specify the arguments that correspond with the parameters. For example `mycorearch1` expects two parameters which are used in the architecture definition:

```
core mycore
{
     architecture = mycorearch1 (1,2);
}
```

### 7.6.3 DEFINING INTERNAL MEMORY AND BUSES

With the **`memory`** keyword you define physical memory that is present on the target board. The memory name is used to identify the memory and does not conflict with other identifiers. It is common to define internal memory (on–chip) in the derivative definition. External memory (off–chip memory) is usually defined in the board specification (See section 7.7.3, *Defining External Memory and Buses*).

- The **`type`** field specifies a memory type:
  - **`rom`**: read only memory
  - **`ram`**: random access memory
  - **`nvram`**: non volatile ram

  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 7.8.2, *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 (0..4): 0 is the fastest, 4 the slowest. The linker uses the relative speed of the memories in such a way, that optimal speed is achieved. The default speed is 1.
- The **`map`** field specifies how this memory maps onto an (internal) bus. Mappings are described in section 7.5.4, *Mappings*.

```
memory mem_name
{
   type = rom;
   mau = 8;
   size = 64k;
   speed = 2;
   map ( map_description );
}
```

With the **bus** keyword you define a bus in a derivative definition. Buses are described in section 7.5.2, *Defining Internal Buses*.

## 7.7 SEMANTICS OF THE BOARD SPECIFICATION

***Keywords in the board specification***

```
processor
   derivative
bus
   mau
   width
   map
memory
   type              reserved  rom  ram  nvram
   mau
   size
   speed
   map

   map
      dest           bus  space
      dest_dbits
      dest_offset
      size
      src_dbits
      src_offset
```

### 7.7.1 DEFINING A PROCESSOR

If you have a target board with multiple processors that have the same derivative, you need to instantiate each individual processor in a processor definition. This information tells the linker which processor has which derivative and enables the linker to distinguish between the present processors.

If you use processors that all have a unique derivative, you may omit the processor definitions. In this case the linker assumes that for each derivative definition in the LSL file there is one processor. The linker uses the derivative name also for the processor.

With the keyword **processor** you define a processor. You can freely choose the processor name. The name is used to refer to it at other places in the LSL file:

```
processor proc_name
{
   processor definition
}
```

### 7.7.2 INSTANTIATING DERIVATIVES

With the keyword **derivative** you tell the linker that the given processor has a certain derivative. The derivative name refers to an existing derivative definition in the same LSL file.

For examples, if you have two processors on your target board (called `myproc_1` and `myproc_2`) that have the same derivative (called `myderiv`), you must instantiate both processors as follows:

```
processor myproc_1
{
     derivative = myderiv;
}

processor myproc_2
{
     derivative = myderiv;
}
```

If the derivative definition has parameters you must specify the arguments that correspond with the parameters. For example `myderiv1` expects two parameters which are used in the derivative definition:

```
processor myproc
{
     derivative = myderiv1 (2,4);
}
```

### 7.7.3 DEFINING EXTERNAL MEMORY AND BUSES

It is common to define external memory (off–chip) and external buses at the global scope (outside any enclosing definition). Internal memory (on–chip memory) is usually defined in the scope of a derivative definition.

With the keyword **memory** you define physical memory that is present on the target board. The memory name is used to identify the memory and does not conflict with other identifiers. If you define memory parts in the LSL file, only the memory defined in these parts is used for placing sections.

If no external memory is defined in the LSL file and if the linker option to allocate memory on demand is set then the linker will assume that all virtual addresses are mapped on physical memory. You can override this behavior by specifying one or more memory definitions.

```
memory mem_name
{
   type = rom;
   mau = 8;
   size = 64k;
   speed = 2;
   map ( map_description );
}
```

For a description of the keywords, see section 7.6.3, *Defining Internal Memory and Buses*.

With the keyword **bus** you define a bus (the combination of data and corresponding address bus). The bus name is used to identify a bus and does not conflict with other identifiers. Bus descriptions at the global scope (outside any definition) define *external* buses. These are buses that are present on the target board.

```
bus bus_name
{
      mau = 8;
      width = 8;
      map ( map_description );
}
```

For a description of the keywords, see section 7.5.2, *Defining Internal Buses*.

You can connect off–chip memory to any derivative: you need to map the off–chip memory to a bus and map that bus on the internal bus of the derivative you want to connect it to.

## 7.8 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
   fill
   ordered
   clustered
   contiguous
   overlay
   allow_cross_references
   load_addr
      mem
   run_addr
      mem
   page
select
heap
   size
stack
   size
reserved
   size
   attributes     r w x
   fill
   alloc_allowed absolute
copytable
section
   size
   attributes     r w x
   fill

if
else
```

### 7.8.1 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.

### 7.8.2 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 7.8.3, *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 **select** keyword you can select one or more sections for the group. You can select a section by name or by attributes. If you select a section by name, you can use a wildcard pattern:

| | |
|---|---|
| "*" | matches with all section names |
| "?" | matches with a single character in the section name |
| "\" | takes the next character literally |
| "[abc]" | matches with a single 'a', 'b' or 'c' character |
| "[a–z]" | matches with any single character in the range 'a' to 'z' |

```
group ( ... )
{
   select ".mysection";
   select "*";
}
```

The first **select** statement selects the section with the name ".mysection". The second **select** statement selects all sections that were not selected yet.

A section is selected by the first **select** statement that matches, in the union of all section layouts for the address space. Global section layouts are processed in the order in which they appear in the LSL file. Internal core architecture section layouts always take precedence over global section layouts.

- The **attributes** field selects all sections that carry (or do not carry) the given attribute. With **+***attribute* you select sections that have the specified attribute set. With **–***attribute* you select sections that do not have the specified attribute set. You can specify one or more of the following attributes:
  - **r** readable sections
  - **w** readable/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);
}
```

Keep in mind that all section selections are restricted to the address space of the section layout in which this group definition occurs.

### *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. 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 the sections in the group. Default the linker uses the attributes of the input sections. The list of available attributes is the same as described above for the selection of 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'. 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 contigous 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 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. With an expression you can specify that the group should be located at the absolute address specified by the expression:

     ```
     group (run_addr = 0xa00f0000)
     ```

     You can use the '**[***offset***]**' variant to locate the group at the given absolute offset in memory:

     ```
     group (run_addr = mem:A[0x1000])
     ```

     A range can be an absolute space address range, written as **[** *expr* **..** *expr* **]**, a complete memory device, written as **mem:***mem_name*, or a memory address range, **mem:***mem_name***[** *expr* **..** *expr* **]**

```
group (run_addr = mem:my_dram)
```

You can use the '|' to specify an address range of more than one physical memory device:

```
group (run_addr = mem:A | mem:B)
```

- The **load_addr** keyword changes the meaning of the section selection in the group: the linker selects the load–time ROM copy of the named section(s) instead of the regular sections. Just like **run_addr** you can specify an absolute address or an address range.

  The **load_addr** keyword itself (without an assignment) specifies that the group's position in the LSL file defines its load–time address.

```
group (load_addr)
select "mydata";  // select ROM copy of mydata:
                  // "[mydata]"
```

The load–time and run–time addresses of a group cannot be set at the same time. If the load–time property is set for a group, the group (only) restricts the positioning at load–time of the group's sections. It is not possible to set the address of a group that has a not–unrestricted parent group.

The properties of the load–time and run–time start address are:

- At run–time, before using an element in an overlay group, the application copies the sections from their load location to their run–time location, but only if these two addresses are different. For non–overlay sections this happens at program start–up.
- The start addresses cannot be set to absolute values for unrestricted groups.
- For non–overlay groups that do not have an overlay parent, the load–time start address equals the run–time start address.
- For any group, if the run–time start address is not set, the linker selects an appropriate address.

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. See also the **page** keyword in section 7.5.3, *Defining Address Spaces*.

```
group (page, ... )
group (page = 3, ...)
```

### 7.8.3 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 **stack** keyword 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 **fixed** keyword.

  ```
  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 7.5.3, *Defining Address Spaces*.

***Heap***

- The **heap** keyword tells the linker to reserve a dynamic memory range for the `malloc()` function. Optionally you can assign a name to the heap section. With the keyword **size** you can change the size for the heap. If the **size** is not specified, the linker uses the size given by the **min_size** argument as defined for the heap in the architecture definition. Normally the linker automatically tries to maximize the size, unless you specified the **fixed** keyword.

```
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 **reserved** keyword tells the linker to create an area or section of a given size. The linker will not locate any other sections in the memory occupied by a reserved section, with some exceptions. Optionally you can assign a name to a reserved section. With the keyword **size** you can specify a size for a given reserved area or section.

```
group ( ... )
{
   reserved "myreserved" ( size = 2k );
}
```

The optional **fill** field contains a bit pattern that the linker writes to all memory addresses that remain unoccupied during the locate process. The result of the expression, or list of expressions, is used as values to write to memory, each in MAU. The first MAU of the fill pattern is always the first MAU in the section.

By default, no sections can overlap with a reserved section. With **alloc_allowed=absolute** sections that are located at an absolute address due to an absolute group restriction can overlap a reserved section.

With the **attributes** field you can set the access type of the reserved section. The linker locates the reserved section in its space with the restrictions that follow from the used attributes, **r**, **w** or **x** or a valid combination of them. The allowed attributes are shown in the following table. A value between < and > in the table means this value is set automatically by the linker.

| Properties set in LSL | | Resulting section properties | | |
|---|---|---|---|---|
| **attributes** | **filled** | **access** | **memory** | **content** |
| x | yes | | <rom> | executable |
| r | yes | r | <rom> | data |
| r | no | r | <rom> | scratch |
| rx | yes | r | <rom> | executable |
| rw | yes | rw | <ram> | data |
| rw | no | rw | <ram> | scratch |
| rwx | yes | rw | <ram> | executable |

```
group ( ... )
{
   reserved "myreserved" ( size = 2k,
               attributes = rw, fill = 0xaa );
}
```

If you do not specify any attributes, the linker will reserve the given number of maus, no matter what type of memory lies beneath. If you do not specify a fill pattern, no section is generated.

The linker creates two labels to mark the begin and end of the section, **__lc_ub_***name* for the start, and **__lc_ue_***name* for the end of the reserved section.

### *Output sections*

- The **section** keyword 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. With the keyword **size** you specify the size of the output section.

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 linker creates two labels to mark the begin and end of the section, **__lc_ub_***name* for the start, and **__lc_ue_***name* for the end of the output section.

***Copy table***

- The **copytable** keyword tells the linker to select a section that is used as *copy–table*. The content of the copy–table is created by the linker. It contains the start address and length of all sections that should be initialized by the startup code.

  The linker creates two labels to mark the begin and end of the section, **__lc_ub_table** for the start, and **__lc_ue_table** for the end of the copy table. The linker generates a copy table when a reference to either of the section labels exists in one of the input object files.

## 7.8.4 CREATING SYMBOLS

You can tell the linker to create symbols before locating by putting assignments in the section layout definition. Symbol names are represented by double–quoted strings. Any string is allowed, but object files may not support all characters for symbol names. You can use two different assignment operators. With the simple assignment operator '**=**', the symbol is created unconditionally. With the '**:=**' operator, the symbol is only created if it already exists as an undefined reference in an object file.

The expression that represents the value to assign to the symbol may contain references to other symbols. If such a referred symbol is a special section symbol, creation of the symbol in the left hand side of the assignment will cause creation of the special section.

```
section_layout
{
   "__lc_bs" := "__lc_ub_stack";
    // when the symbol __lc_bs occurs in the object
    // file, the linker allocates space for the stack
}
```

### 7.8.5 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 ( size_of ( sect:.mysection ) < 2k )
      select ".mysection";
   else
      select ".othersection";
}
```

CHAPTER 8

# MISRA C RULES

TASKING

# 8 CHAPTER

### *Supported and unsupported MISRA C rules*

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.

1. The code shall conform to standard C, without language extensions

* 2. Other languages should only be used with an interface standard

3. Inline assembly is only allowed in dedicated C functions

* 4. Provision should be made for appropriate run–time checking

5. Only use characters and escape sequences defined by ISO C

* 6. Character values shall be restricted to a subset of ISO 106460–1

7. Trigraphs shall not be used

8. Multibyte characters and wide string literals shall not be used

9. Comments shall not be nested

10. 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. Identifiers shall not rely on significance of more than 31 characters

12. The same identifier shall not be used in multiple name spaces

13. Specific–length typedefs should be used instead of the basic types

14. Use 'unsigned char' or 'signed char' instead of plain 'char'

* 15. Floating point implementations should comply with a standard

16. 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. "typedef" names shall not be reused

18. 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. Octal constants (other than zero) shall not be used

20. All object and function identifiers shall be declared before use

21. Identifiers shall not hide identifiers in an outer scope

22. Declarations should be at function scope where possible

* 23. All declarations at file scope should be static where possible

24. Identifiers shall not have both internal and external linkage

* 25. Identifiers with external linkage shall have exactly one definition

26. Multiple declarations for objects or functions shall be compatible

* 27. External objects should not be declared in more than one file

28. The "register" storage class specifier should not be used

29. The use of a tag shall agree with its declaration

30. 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. Braces shall be used in the initialization of arrays and structures

32. Only the first, or all enumeration constants may be initialized

33. The right hand operand of && or || shall not contain side effects

34. The operands of a logical && or || shall be primary expressions

35. Assignment operators shall not be used in Boolean expressions

36. Logical operators should not be confused with bitwise operators

37. Bitwise operations shall not be performed on signed integers

38. 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. The unary minus shall not be applied to an unsigned expression

40. ”sizeof” should not be used on expressions with side effects

* 41. The implementation of integer division should be documented

42. The comma operator shall only be used in a ”for” condition

43. Don’t use implicit conversions which may result in information loss

44. Redundant explicit casts should not be used

45. Type casting from any type to or from pointers shall not be used

46. 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. No dependence should be placed on operator precedence rules

48. Mixed arithmetic should use explicit casting

49. Tests of a (non–Boolean) value against 0 should be made explicit

50. F.P. variables shall not be tested for exact equality or inequality

51. Constant unsigned integer expressions should not wrap–around

52. There shall be no unreachable code

53. All non–null statements shall have a side–effect

54. A null statement shall only occur on a line by itself

55. Labels should not be used

56. The ”goto” statement shall not be used

57. The ”continue” statement shall not be used

58. The ”break” statement shall not be used (except in a ”switch”)

59. An ”if” or loop body shall always be enclosed in braces
60. All ”if”, ”else if” constructs should contain a final ”else”
61. Every non–empty ”case” clause shall be terminated with a ”break”
62. All ”switch” statements should contain a final ”default” case
63. A ”switch” expression should not represent a Boolean case
64. Every ”switch” shall have at least one ”case”
65. Floating point variables shall not be used as loop counters
66. 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. Iterator variables should not be modified in a ”for” loop
68. Functions shall always be declared at file scope
69. Functions with variable number of arguments shall not be used
70. 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. Function prototypes shall be visible at the definition and call
72. The function prototype of the declaration shall match the definition
73. Identifiers shall be given for all prototype parameters or for none
74. Parameter identifiers shall be identical for declaration/definition
75. Every function shall have an explicit return type
76. Functions with no parameters shall have a ”void” parameter list
77. An actual parameter type shall be compatible with the prototype
78. The number of actual parameters shall match the prototype
79. The values returned by ”void” functions shall not be used

80. Void expressions shall not be passed as function parameters

81. ”const” should be used for reference parameters not modified

82. A function should have a single point of exit

83. Every exit point shall have a ”return” of the declared return type

84. For ”void” functions, ”return” shall not have an expression

85. Function calls with no parameters should have empty parentheses

86. If a function returns error information, it should be tested

    A violation is reported when a the return value of a function is ignored.

87. #include shall only be preceded by another directives or comments

88. Non–standard characters shall not occur in #include directives

89. #include shall be followed by either <filename> or ”filename”

90. Plain macros shall only be used for constants/qualifiers/specifiers

91. Macros shall not be #define’d and #undef’d within a block

92. #undef should not be used

93. A function should be used in preference to a function–like macro

94. A function–like macro shall not be used without all arguments

95. Macro arguments shall not contain pre–preprocessing directives

    A violation is reported when the first token of an actual macro argument is *’#’*.

96. Macro definitions/parameters should be enclosed in parentheses

97. Don’t use undefined identifiers in pre–processing directives

98. A macro definition shall contain at most one # or ## operator

99. 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. ”defined” shall only be used in one of the two standard forms

101. Pointer arithmetic should not be used

102. 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. 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. Non–constant pointers to functions shall not be used

105. Functions assigned to the same pointer shall be of identical type

106. Automatic address may not be assigned to a longer lived object

107. 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. All struct/union members shall be fully specified

109. Overlapping variable storage shall not be used

A violation is reported for every ’union’ declaration.

110. 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. Bit fields shall have type ”unsigned int” or ”signed int”

112. Bit fields of type ”signed int” shall be at least 2 bits long

113. All struct/union members shall be named

114. Reserved and standard library names shall not be redefined

115. Standard library function names shall not be reused

* 116. Production libraries shall comply with the MISRA C restrictions

* 117. The validity of library function parameters shall be checked

118. Dynamic heap memory allocation shall not be used
119. The error indicator ”errno” shall not be used
120. The macro ”offsetof” shall not be used
121. <locale.h> and the ”setlocale” function shall not be used
122. The ”setjmp” and ”longjmp” functions shall not be used
123. The signal handling facilities of <signal.h> shall not be used
124. The <stdio.h> library shall not be used in production code
125. The functions atof/atoi/atol shall not be used
126. The functions abort/exit/getenv/system shall not be used
127. The time handling functions of library <time.h> shall not be used

* = Not supported by the TASKING C compiler

See also section 5.7, *C Code Checking: MISRA C*, in Chapter *Using the Compiler* of the *User's Guide*.

INDEX

# INDEX

# INDEX

# Symbols



# A

INDEX

# B



# C

# D



# E

# F



# G

## H



## I

## J



## L

INDEX

# M

# N



# O



# P

# Q



# R



# S

## T



## U



## V

# W

INDEX