# TASKING VX-toolset for MCS User Guide

MA162-800 (v1.3) July 12, 2013

# Table of Contents

# Chapter 1. Assembly Language

This chapter describes the most important aspects of the TASKING assembly language for the Multi Channel Sequencer (MCS). For a complete overview of the MCS, refer to the Generic Timer Module (GTM) chapter in the *AURIX TC27x 32-Bit Single-Chip Microcontroller Target Specification* [V2.4, 2011-08, Infineon].

## 1.1. Assembly Syntax

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

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

The syntax of an assembly statement is:

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

*label*

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

*number* is a number ranging from 1 to 255. This type of label is called a *numeric label* or *local label*. To refer to a numeric label, you must put an **n** (next) or **p** (previous) immediately after the label. This is required because the same label number may be used repeatedly.

Examples:

```
   LAB1:   ; This label is followed by a colon and
           ; can be prefixed by whitespace
LAB1       ; This label has to start at the beginning
           ; of a line
1: jmp 1p  ; This is an endless loop
           ; using numeric labels
```

*instruction*

An instruction consists of a mnemonic and zero, one or more operands. It must not start in the first column.

Operands are described in Section 1.3, *Operands of an Assembly Instruction*. The instructions are described in the Target Specification Manual.

*directive* With directives you can control the assembler from within the assembly source. Except for preprocessing directives, these must not start in the first column. Directives are described in Section 1.9, *Assembler Directives and Controls*.

*macro_call* A call to a previously defined macro. It must not start in the first column. See Section 1.10, *Macro Operations*.

*comment* Comment, preceded by a ; (semicolon).

You can use empty lines or lines with only comments.

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

```
$control
```

For more information on controls see Section 1.9, *Assembler Directives and Controls*.

## 1.2. Assembler Significant Characters

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

Some characters have a special meaning to the assembler. Special characters associated with expression evaluation are described in Section 1.6.3, *Expression Operators*. Other special assembler characters are:

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

## 1.3. Operands of an Assembly Instruction

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

| Operand | Description |
|---|---|
| *symbol* | A symbolic name as described in Section 1.4, *Symbol Names*. Symbols can also occur in expressions. |
| *register* | Any valid register as listed in Section 1.5, *Registers*. |
| *expression* | Any valid expression as described in Section 1.6, *Assembly Expressions*. |
| *address* | A combination of *expression*, *register* and *symbol*. |

### Addressing modes

The MCS assembly language has several addressing modes. These addressing modes are used for indirect memory addressing or indirect ARU addressing. For details see the *AURIX TC27x 32-Bit Single-Chip Microcontroller Target Specification* [V2.4, 2011-08, Infineon].

## 1.4. Symbol Names

### User-defined symbols

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

### Predefined preprocessor symbols

These symbols start and end with two underscore characters, __*symbol*__, and you can use them in your assembly source to create conditional assembly. See Section 1.4.1, *Predefined Preprocessor Symbols*.

### Labels

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

### Reserved symbols

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

### Examples

Valid symbol names:

```
loop_1
ENTRY
a_B_c
_aBC
```

Invalid symbol names:

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

### 1.4.1. Predefined Preprocessor Symbols

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

| Symbol | Description |
|---|---|
| __ASMCS__ | Identifies the assembler. You can use this symbol to flag parts of the source which must be recognized by the **asmcs** assembler only. It expands to 1. |
| __BUILD__ | Identifies the build number of the assembler, composed of decimal digits for the build number, three digits for the major branch number and three digits for the minor branch number. For example, if you use build 1.22.1 of the assembler, __BUILD__ expands to 1022001. If there is no branch number, the branch digits expand to zero. For example, build 127 results in 127000000. |
| __REVISION__ | Expands to the revision number of the assembler. Digits are represented as they are; characters (for prototypes, alphas, betas) are represented by -1. Examples: v1.0r1 -> 1, v1.0rb -> -1 |
| __TASKING__ | Identifies the assembler as a TASKING assembler. Expands to 1 if a TASKING assembler is used. |
| __VERSION__ | Identifies the version number of the assembler. For example, if you use version 2.1r1 of the assembler, __VERSION__ expands to 2001 (dot and revision number are omitted, minor version number in 3 digits). |

#### Example

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

## 1.5. Registers

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

```
R0  .. R7   (general purpose registers)
STA         (status register)
ACB         (ARU control bit register)
CTRG        (clear trigger bits register)
STRG        (set trigger bits register)
TBU_TS0     (TBU timestamp TS0 register)
TBU_TS1     (TBU timestamp TS1 register)
TBU_TS2     (TBU timestamp TS2 register)
MHB         (memory high byte register)
```

## 1.6. Assembly Expressions

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

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

Expressions follow the conventional rules of algebra and boolean arithmetic.

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

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

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

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

- *numeric constant*
- *string*
- *symbol*
- *expression binary_operator expression*
- *unary_operator expression*
- **(***expression***)**
- *function call*

All types of expressions are explained in separate sections.

## 1.6.1. Numeric Constants

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

| Base | Description | Example |
|---|---|---|
| Binary | A **0b** or **0B** prefix followed by binary digits (0,1). | `0B1101`<br>`0b11001010` |
| Hexadecimal | A **0x** or **0X** prefix followed by hexadecimal digits (0-9, A-F, a-f). | `0X12FF`<br>`0x45`<br>`0xfa10` |
| Decimal integer | Decimal digits (0-9). | `12`<br>`1245` |

## 1.6.2. Strings

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

Strings constants in expressions are evaluated to a number (each character is replaced by its ASCII value). Strings in expressions can have a size of up to 4 characters or less depending on the operand of an instruction or directive; any subsequent characters in the string are ignored. In this case the assembler issues a warning. Null strings have a value of 0.

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

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

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

### Examples

```
'ABCD'              ; (0x41424344)
'''79'              ; to enclose a quote double it
"A\"BC"             ; or to enclose a quote escape it
'AB'+1              ; (0x4143) string used in expression
''                  ; null string
.word 'abcdef'      ; (0x64636261) 'ef' are ignored
                    ; warning: string value truncated
'abc'++'de'         ; you can concatenate
                    ; two strings with the '++' operator.
                    ; This results in 'abcde'
['TASKING',0,4]     ; results in the substring 'TASK'
```

### 1.6.3. Expression Operators

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

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

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

| Type | Operator | Name | Description |
|---|---|---|---|
| | ( ) | parenthesis | Expressions enclosed by parenthesis are evaluated first. |
| Unary | + | plus | Returns the value of its operand. |
| | - | minus | Returns the negative of its operand. |
| | ~ | one's complement | Integer only. Returns the one’s complement of its operand. It cannot be used with a floating-point operand. |
| | ! | logical negate | Returns 1 if the operands' value is 0; otherwise 0. For example, if `buf` is 0 then `!buf` is 1. If `buf` has a value of 1000 then `!buf` is 0. |
| Arithmetic | * | multiplication | Yields the product of its operands. |
| | / | division | Yields the quotient of the division of the first operand by the second. For integer operands, the divide operation produces a truncated integer result. |
| | % | modulo | Integer only. This operator yields the remainder from the division of the first operand by the second. |
| | + | addition | Yields the sum of its operands. |
| | - | subtraction | Yields the difference of its operands. |
| Shift | << | shift left | Integer only. Causes the left operand to be shifted to the left (and zero-filled) by the number of bits specified by the right operand. |
| | >> | shift right | Integer only. Causes the left operand to be shifted to the right by the number of bits specified by the right operand. The sign bit will be extended. |

| Type | Operator | Name | Description |
|---|---|---|---|
| Relational | < | less than | Returns an integer 1 if the indicated condition is TRUE or an integer 0 if the indicated condition is FALSE.<br>For example, if D has a value of 3 and E has a value of 5, then the result of the expression `D<E` is 1, and the result of the expression `D>E` is 0.<br>Use tests for equality involving floating-point values with caution, since rounding errors could cause unexpected results. |
|  | <= | less than or equal |  |
|  | > | greater than |  |
|  | >= | greater than or equal |  |
|  | == | equal |  |
|  | != | not equal |  |
| Bit and Bitwise | & | AND | Integer only. Yields the bitwise AND function of its operand. |
|  | \| | OR | Integer only. Yields the bitwise OR function of its operand. |
|  | ^ | exclusive OR | Integer only. Yields the bitwise exclusive OR function of its operands. |
| Logical | && | logical AND | Returns an integer 1 if both operands are non-zero; otherwise, it returns an integer 0. |
|  | \|\| | logical OR | Returns an integer 1 if either of the operands is non-zero; otherwise, it returns an integer 1 |

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

## 1.7. Working with Sections

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

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

For more information about locating sections see Section 3.6.8, *The Section Layout Definition: Locating Sections*.

### Section definition

Sections are defined with the `.SDECL` directive and have a name. A section may have attributes to instruct the linker to place it on a predefined starting address, or that it may be overlaid with another section.

```
.SDECL "name",  type  [, attribute ]...  [AT address]
```

See the description of the `.SDECL` directive for a complete description of all possible attributes.

### Section activation

Sections are defined once and are activated with the `.SECT` directive.

```
.SECT "name"
```

The linker will check between different modules and emits an error message if the section attributes do not match. The linker will also concatenate all matching section definitions into one section. So, all "code" sections will be linked into one big "code" chunk which will be located in one piece. A `.SECT` directive referring to an earlier defined section is called a *continuation*. Only the name can be specified.

### Examples

```
.SDECL  ".mcstext.code",CODE
.SECT   ".mcstext.code"
```

Defines and activates a relocatable section in CODE memory. Other parts of this section, with the same name, may be defined in the same module or any other module. Other modules should use the same `.SDECL` statement. When necessary, it is possible to give the section an absolute starting address.

```
.SDECL  ".mcsdata.data", data at 0x100
.SECT   ".mcsdata.data"
```

Defines and activates an absolute section named `.mcsdata.data` starting at address 0x100.

## 1.8. Built-in Assembly Functions

The TASKING assembler has several built-in functions to support string comparison and macro testing. You can use functions as terms in any expression.

### Syntax of an assembly function

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

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

The names of assembly functions are case insensitive.

### Overview of mathematical functions

| Function | Description |
|---|---|
| `@SGN(`*expr*`)` | Returns the sign of an expression as -1, 0 or 1 |

## Overview of conversion functions

| Function | Description |
|---|---|
| `@RVB(`*expr*`[,`*exprN*`])` | Reverse order of bits in field |

## Overview of string functions

| Function | Description |
|---|---|
| `@CAT(`*str1*`,`*str2*`)` | Concatenate *str1* and *str2* |
| `@LEN(`*string*`)` | Length of string |
| `@POS(`*str1*`,`*str2*`[,`*start*`])` | Position of *str2* in *str1* |
| `@SCP(`*str1*`,`*str2*`)` | Compare *str1* with *str2* |
| `@SUB(`*str*`,`*expr1*`,`*expr2*`)` | Return substring |

## 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 macro is defined |
| `@MXP()` | Test if macro expansion is active |

## Overview of address calculation functions

| Function | Description |
|---|---|
| `@LSB(`*expr*`)` | Least significant byte of the expression |
| `@MSB(`*expr*`)` | Most significant byte of the expression |

## Overview of assembler mode functions

| Function | Description |
|---|---|
| `@CPU('`*cpu*`')` | Test if CPU type is selected |
| `@DEF('`*symbol*`'` \| *symbol*`)` | Returns 1 if symbol has been defined |
| `@EXP(`*expr*`)` | Expression check |
| `@INT(`*expr*`)` | Integer check |
| `@LST()` | LIST control flag value |

## Detailed Description of Built-in Assembly Functions

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

Returns integer 1 if the macro argument represented by *symbol* or *expression* is present, 0 otherwise.

You can specify the argument with a *symbol* name (the name of a macro argument enclosed in single quotes) or with *expression* (the ordinal number of the argument in the macro formal argument list). If you use this function when macro expansion is not active, the assembler issues a warning.

Example:

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

### @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('TASK','ING')"   ;ID = 'TASKING'
```

### @CNT()

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

Example:

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

### @CPU(*string*)

Returns integer 1 if *string* corresponds to the selected CPU type; 0 otherwise. See also assembler option **--cpu** (Select CPU).

Example:

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

Returns 1 if *symbol* has been defined, 0 otherwise. *symbol* can be any symbol or label not associated with a `.MACRO` or `.SDECL` 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 @DEFINED('ANGLE')          ;is symbol ANGLE defined?
.IF @DEFINED(ANGLE)            ;does label ANGLE exist?
```

## @EXP(*expression*)

Returns 0 if the evaluation of *expression* would normally result in an error. Returns 1 if the expression can be evaluated correctly. With the @EXP function, you prevent the assembler from generating an error if the expression contains an error. No test is made by the assembler for warnings. The expression may be relative or absolute.

Example:

```
.IF  !@EXP(3/0)       ;Do the IF on error
                      ;assembler generates no error

.IF  !(3/0)           ;assembler generates an error
```

## @INT(*expression*)

Returns integer 1 if *expression* has an integer result; otherwise, it returns a 0. The expression may be relative or absolute.

Example:

```
.IF  @INT(TERM)     ;Test if result is an integer
```

## @LEN(*string*)

Returns the length of *string* as an integer.

Example:

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

## @LSB(*expression*)

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

Example:

```
VAR1   .SET @LSB(0x34)        ;VAR1 = 0x34
VAR2   .SET @LSB(0x1234)      ;VAR2 = 0x34
VAR3   .SET @LSB(0x654321)    ;VAR3 = 0x21
```

## @LST()

Returns the value of the `$LIST ON/OFF` control flag as an integer. Whenever a `$LIST ON` control is encountered in the assembler source, the flag is incremented; when a `$LIST OFF` control is encountered, the flag is decremented.

Example:

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

## @MAC(*symbol*)

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

Example:

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

## @MSB(*expression*)

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

Example:

```
VAR1   .SET @MSB(0x34)           ;VAR1 = 0x00
VAR2   .SET @MSB(0x1234)         ;VAR2 = 0x12
VAR3   .SET @MSB(0x654321)       ;VAR3 = 0x43
```

## @MXP()

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

With *start* you can specify the starting position of the search. If you do not specify start, the search is started from the beginning of *string1*. Note that the first position in a string is position 0.

Example:

```
ID1  .EQU  @POS('TASKING','ASK')  ; ID1 = 1
ID2  .EQU  @POS('ABCDABCD','B',2) ; ID2 = 5
ID3  .EQU  @POS('TASKING','BUG')  ; ID3 = 7
```

## @RVB(*expression1*,*expression2*)

Reverse the order of bits in *expression1* delimited by the number of bits in *expression2*. If *expression2* is omitted the field is bounded by the target word size. Both expressions must be 16-bit integer values.

Example:

```
VAR1 .SET @RVB(0x200)   ;reverse all bits, VAR1=0x40
VAR2 .SET @RVB(0xB02)   ;reverse all bits, VAR2=0x40D0
```

```
VAR3 .SET @RVB(0xB02,2) ;reverse bits 0 and 1,
                        ;VAR3=0xB01
```

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

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

Example:

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

### @SGN(*expression*)

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

Example:

```
VAR1  .SET @SGN(-1.2e-92)   ;VAR1 = -1
VAR2  .SET @SGN(0)          ;VAR2 =  0
VAR3  .SET @SGN(28.382)     ;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('TASKING',3,4)"  ;ID = 'KING'
```

## 1.9. Assembler Directives and Controls

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

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

  The following directives fall under this group:

  - Assembly control directives
  - Symbol definition and section directives
  - Data definition / Storage allocation directives

  - High Level Language (HLL) directives

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

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

  The following controls are available:

  - Assembly listing controls

  - Miscellaneous controls

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

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

## 1.9.1. Assembler Directives

### Overview of assembly control directives

| Directive | Description |
|---|---|
| `.COMMENT` | Start comment lines. You cannot use this directive in .IF/.ELSE/.ENDIF constructs and .MACRO/.DUP definitions. |
| `.END` | Indicates the end of an assembly module |
| `.FAIL` | Programmer generated error message |
| `.INCLUDE` | Include file |
| `.MESSAGE` | Programmer generated message |
| `.WARNING` | Programmer generated warning message |

### Overview of symbol definition and section directives

| Directive | Description |
|---|---|
| `.EQU` | Set permanent value to a symbol |
| `.EXTERN` | Import global section symbol |
| `.GLOBAL` | Declare global section symbol |

| Directive | Description |
|---|---|
| .LOCAL | Declare local section symbol |
| .ORG | Initialize memory space and location counters to create a nameless section |
| .SDECL | Declare a section with name, type and attributes |
| .SECT | Activate a declared section |
| .SET | Set temporary 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 a symbol as 'weak' |

## Overview of data definition / storage allocation directives

| Directive | Description |
|---|---|
| .ALIGN | Align location counter |
| .ASCII, .ASCIIZ | Define ASCII string without / with ending NULL byte |
| .SPACE | Define storage (32 bits) |
| .WORD | Define word (32 bits) |

## Overview of macro preprocessor directives

| Directive | Description |
|---|---|
| .DEFINE | Define substitution string |
| .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 |
| .IF, .ELIF, .ELSE | Conditional assembly directive |
| .ENDIF | End of conditional assembly directive |
| .EXITM | Exit macro |
| .MACRO, .ENDM | Define macro |
| .PMACRO | Undefine (purge) macro |
| .UNDEF | Undefine .DEFINE symbol |

## .ALIGN

### Syntax

```
.ALIGN expression
```

### Description

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

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

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

A label is not allowed before this directive.

### Example

```
.sdecl '.mcstext.code',code
.sect  '.mcstext.code'
.ALIGN 16    ; the assembler aligns
instruction  ; this instruction at 16 MAUs and
             ; fills the 'gap' with NOP instructions.

.sdecl '.mcstext.code',code
.sect  '.mcstext.code'
.ALIGN 12    ; WRONG: not a power of two, the
instruction  ; assembler aligns this instruction at
             ; 16 MAUs and issues a warning.
```

## .ASCII, .ASCIIZ

### Syntax

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

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

### Description

With the `.ASCII` or `.ASCIIZ` directive the assembler allocates and initializes memory for each *string* argument. The last word will be padded with 0x00. Use commas to separate multiple strings.

There is only a difference between `.ASCII` and `.ASCIIZ` when the *string* has a size that is a multiple of four characters. In that case the `.ASCIIZ` directive adds a word with all zeros. The "z" in `.ASCIIZ` stands for "zero"

### Example

```
STRING:  .ASCII  "Hello world"   ; = 0x48656C6C
                                 ;   0x6F20776F
                                 ;   0x726C6400
STRINGZ: .ASCIIZ "Hello world"   ; = 0x48656C6C
                                 ;   0x6F20776F
                                 ;   0x726C6400
STR4:    .ASCII  "Four"          ; = 0x466F7572
STR8:    .ASCIIZ "Four"          ; = 0x466F7572
                                 ;   0x00000000
```

### Related Information

`.SPACE` (Define Storage)

# .COMMENT

## Syntax

```
.COMMENT  delimiter
.
.
delimiter
```

## Description

With the .COMMENT directive 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.
          *
```

## .DEFINE

### Syntax

```
.DEFINE symbol  string
```

### Description

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

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

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

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

A label is not allowed before this directive.

### Example

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

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

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

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

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

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

### Related Information

`.UNDEF` (Undefine a .DEFINE symbol)

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

## .DUP, .ENDM

### Syntax

```
[label:] .DUP expression
    ....
    .ENDM
```

### Description

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

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

### Example

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

```
.DUP 3
.WORD 10  ; assembly source lines
.ENDM
```

### Related Information

`.DUPA, .ENDM` (Duplicate sequence with arguments)

`.DUPC, .ENDM` (Duplicate sequence with characters)

`.DUPF, .ENDM` (Duplicate sequence in loop)

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

## .DUPA, .ENDM

### Syntax

```
[label:] .DUPA formal_arg,argument[,argument]...
    ....
    .ENDM
```

### Description

With the `.DUPA`/`.ENDM` directive 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 directive processing.

### Example

Consider the following source input statements,

```
  .DUPA  VALUE,12,,32,34
  .WORD  VALUE
  .ENDM
```

This is expanded as follows:

```
  .WORD  12
  .WORD  VALUE  ; results in a warning
  .WORD  32
  .WORD  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, .ENDM` (Duplicate sequence of source lines)

`.DUPC, .ENDM` (Duplicate sequence with characters)

`.DUPF, .ENDM` (Duplicate sequence in loop)

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

## .DUPC, .ENDM

### Syntax

```
[label:] .DUPC formal_arg,string
    ....
    .ENDM
```

### Description

With the `.DUPC/.ENDM` directive 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 directive processing.

### Example

Consider the following source input statements,

```
  .DUPC   VALUE,'123'
  .WORD   VALUE
  .ENDM
```

This is expanded as follows:

```
  .WORD   1
  .WORD   2
  .WORD   3
```

### Related Information

`.DUP, .ENDM` (Duplicate sequence of source lines)

`.DUPA, .ENDM` (Duplicate sequence with arguments)

`.DUPF, .ENDM` (Duplicate sequence in loop)

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

## .DUPF, .ENDM

### Syntax

```
[label:] .DUPF formal_arg,[start],end[,increment]
    ....
    .ENDM
```

### Description

With the `.DUPF`/`.ENDM` directive 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 directive processing.

### Example

Consider the following source input statements,

```
  .DUPF   NUM,0,7
  .WORD   NUM
  .ENDM
```

This is expanded as follows:

```
  .WORD   0
  .WORD   1
  .WORD   2
  .WORD   3
  .WORD   4
  .WORD   5
  .WORD   6
  .WORD   7
```

## Related Information

`.DUP, .ENDM` (Duplicate sequence of source lines)

`.DUPA, .ENDM` (Duplicate sequence with arguments)

`.DUPC, .ENDM` (Duplicate sequence with characters)

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

## .END

### Syntax

**`.END`**

### Description

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

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

The assembler does not allow a label with this directive.

### Example

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

### Related Information

-

## .EQU

### Syntax

```
symbol .EQU expression
```

### Description

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

### Example

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

```
MYSYMBOL .EQU  0x4000
```

You cannot redefine the symbol `MYSYMBOL` after this.

### Related Information

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

## .EXITM

### Syntax

```
.EXITM
```

### Description

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

A label is not allowed before this directive.

### Example

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

### Related Information

`.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)

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

## .EXTERN

### Syntax

**.EXTERN** *symbol*[,*symbol*]...

### Description

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

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

A label is not allowed with this directive.

### Example

```
.EXTERN AA,CC,DD      ;defined elsewhere
.sdecl ".mcstext.code", code
.sect  ".mcstext.code"
.
.
movl R3,AA    ; AA is used here
.
```

### Related Information

.GLOBAL (Declare global section symbol)

.LOCAL (Declare local section symbol)

## .FAIL

### Syntax

```
.FAIL {str|exp}[,{str|exp}]...
```

### Description

With the `.FAIL` directive you tell the assembler to print an error message to `stderr` during the assembling process.

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

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.

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

A label is not allowed with this directive.

### Example

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

This results in the error:

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

### Related Information

`.MESSAGE` (Programmer generated message)

`.WARNING` (Programmer generated warning)

## .GLOBAL

### Syntax

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

### Description

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

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

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

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

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

The assembler does not allow a label with this directive.

### Example

```
      .sdecl  '.mcsdata.data', data
      .sect   '.mcsdata.data'
      .GLOBAL  LOOPA  ; LOOPA will be globally
                      ; accessible by other modules
LOOPA .EQU 1          ; definition of symbol LOOPA
```

### Related Information

`.EXTERN` (Import global section symbol)

`.LOCAL` (Declare local section symbol)

## .IF, .ELIF, .ELSE, .ENDIF

### Syntax

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

### Description

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

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

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

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

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

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

A label is not allowed with this directive.

### Example

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

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

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

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

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

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

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

# .INCLUDE

## Syntax

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

## Description

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

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

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

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

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

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

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

4. The default include directory in the installation directory.

The assembler does not allow a label with this directive.

## Example

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

## .LOCAL

### Syntax

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

### Description

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

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

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

The assembler does not allow a label with this directive.

### Example

```
      .SDECL  ".mcsdata.data",DATA
      .SECT   ".mcsdata.data"
      .LOCAL  LOOPA    ; LOOPA is local to this section

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

### Related Information

`.EXTERN` (Import global section symbol)

`.GLOBAL` (Declare global section symbol)

## .MACRO, .ENDM

### Syntax

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

### Description

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

The definition of a macro consists of three parts:

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

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

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

You can use the following operators in macro definition statements:

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

### Example

The macro definition:

```
CONST24 .MACRO  reg,value                  ;header
        movl    reg,value                  ;body
        .ENDM                              ;terminator
```

The macro call:

```
    .SDECL   ".mcstext.code",code
    .SECT    ".mcstext.code"
    CONST24  r5,0x123456
```

The macro expands as follows:

```
    movl    r5,0x123456
```

### Related Information

Section 1.10, *Macro Operations*

`.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)

`.PMACRO` (Undefine macro)

`.DEFINE` (Define a substitution string)

## .MESSAGE

### Syntax

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

### Description

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

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

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

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

A label is not allowed with this directive.

### Example

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

`.WARNING` (Programmer generated warning)

## .ORG

### Syntax

```
.ORG [abs-loc][,sect_type][,attribute]...
```

### Description

With the `.ORG` directive you can specify an absolute location (*abs_loc*) in memory of a section. This is the same as a `.SDECL`/`.SECT` without a section name.

This directive uses the following arguments:

| | |
|---|---|
| *abs-loc* | Initial value to assign to the run-time location counter. *abs-loc* must be an absolute expression. If *abs_loc* is not specified, then the value is zero. |
| *sect_type* | An optional section type: code or data |
| *attribute* | An optional section attribute: init, noread, noclear, max, rom, group(*string*), cluster(*string*), protect |

For more information about the section types and attributes see the assembler directive `.SDECL`.

The section type and attributes are case insensitive. A label is not allowed with this directive.

### Example

```
; define a section at location 100 decimal
  .org   100

; define a relocatable nameless section
  .org

; define a relocatable data section
  .org   ,data

; define a data section at 0x8000
  .org   0x8000,data
```

### Related Information

`.SDECL` (Declare section name and attributes)

`.SECT` (Activate a declared section)

## .PMACRO

### Syntax

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

### Description

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

The assembler does not allow a label with this directive.

### Example

```
.PMACRO MAC1,MAC2
```

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

### Related Information

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

## .SDECL

### Syntax

```
.SDECL "name",type[,attribute]... [AT address]
```

### Description

With the `.SDECL` 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.

The *name* specifies the name of the section. The *type* operand specifies the section's type and must be one of:

| Type | Description |
|---|---|
| CODE | Code section. |
| DATA | Data section. |
| DEBUG | Debug section. |

The section type and attributes are case insensitive.

The defined *attribute*s are:

| Attribute | Description | Allowed on type |
|---|---|---|
| AT *address* | Locate the section at the given *address*. | CODE, DATA |
| CLEAR | Sections are zeroed at startup. | DATA |
| CLUSTER( '*name*' ) | Cluster code sections with companion debug sections. Used by the linker during removal of unreferenced sections. The name must be unique for this module (not for the application). | CODE, DATA, DEBUG |
| INIT | Defines that the section contains initialization data, which is copied from ROM to RAM at program startup. | CODE, DATA |
| NOCLEAR | Sections are not zeroed at startup. This is a default attribute for data sections. This attribute is only useful with BSS sections, which are cleared at startup by default. | DATA |
| NOINIT | Defines that the section contains no initialization data. | CODE, DATA |
| NOREAD | Defines that the section can be executed from but not read. | CODE |
| PROTECT | Tells the linker to exclude a section from unreferenced section removal and duplicate section removal. | CODE, DATA |
| ROM | Section contains data to be placed in ROM. This ROM area is not executable. | CODE, DATA |

### Section names

The *name* of a section can have a special meaning for locating sections. The name of code sections should always start with "`.mcstext`". The name of data sections should always start with "`.mcsdata`".

### Example

```
.sdecl  ".mcstext.code", code   ; declare code section
.sect   ".mcstext.code"         ; activate section

.sdecl  ".mcsdata.data", data   ; declare data section
.sect   ".mcsdata.data"         ; activate section

.sdecl  ".mcsdata.abssec", data at 0x100
                                ; absolute section
.sect   ".mcsdata.abssec"       ; activate section
```

### Related Information

`.SECT` (Activate a declared section)

## .SECT

### Syntax

```
.SECT "name" [,RESET]
```

### Description

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

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

### Example

```
.sdecl  ".mcsdata.data", data   ; declare data section
.sect   ".mcsdata.data"         ; activate section
```

### Related Information

`.SDECL` (Declare section name and attributes)

## .SET

### Syntax

```
symbol  .SET  expression

        .SET  symbol  expression
```

### Description

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

The `.SET` directive is useful in establishing temporary or reusable counters within macros. *expression* must be absolute and forward references are allowed.

### Example

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

### Related Information

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

## .SIZE

### Syntax

```
.SIZE  symbol,expression
```

### Description

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

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

### Example

```
_MCS_str:  .type   object   ; object _MCS_str
       .size   _MCS_str,16  ; size of object
       .word   80
       .word   67
       .word   80
       .word   0
```

### Related Information

`.TYPE` (Set symbol type)

## .SPACE

### Syntax

`[`*`label`*`:] `**`.SPACE`**` `*`expression`*

### Description

The `.SPACE` directive reserves a block in memory. The reserved block of memory is not initialized to any value.

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

The *expression* specifies the number of words to be reserved, and how much the location counter will advance. The expression must evaluate to an integer greater than zero and cannot contain any forward references (symbols that have not yet been defined).

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

### Example

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

```
        .sdecl  ".mcsdata.data", data
        .sect   ".mcsdata.data"
uninit  .SPACE  12     ; Sample buffer
```

### Related Information

`.WORD` (Define a constant word)

## .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

```
_MCS_Afunc:  .type   func
```

### Related Information

`.SIZE` (Set symbol size)

## .UNDEF

### Syntax

```
.UNDEF symbol
```

### Description

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

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

The assembler does not allow a label with this directive.

### Example

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

```
.UNDEF LEN
```

### Related Information

`.DEFINE` (Define a substitution string)

## .WARNING

### Syntax

```
.WARNING {str|exp}[,{str|exp}]...
```

### Description

With the `.WARNING` directive you tell the assembler to print a warning message to `stderr` during the assembling process.

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

The total warning count will be incremented as with any other warning. The `.WARNING` directive is for example useful in combination with conditional assembly to indicate which part is assembled. The assembling process proceeds normally after the message has been printed.

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

### Example

```
.WARNING  'Parameter out of range'
```

This results in the warning:

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

### Related Information

`.FAIL` (Programmer generated error)

`.MESSAGE` (Programmer generated message)

## .WEAK

### Syntax

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

### Description

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

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

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

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

### Example

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

### Related Information

`.EXTERN` (Import global section symbol)

`.GLOBAL` (Declare global section symbol)

## .WORD

### Syntax

[*label*:] **.WORD** *argument*[,*argument*]...

### Description

With the `.WORD` directive the assembler allocates and initializes one word (32 bits) of memory for each *argument*.

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

An *argument* can be a single- or multiple-character string constant, an expression or empty.

Multiple arguments are stored in sets of four bytes. One or more arguments can be null (indicated by two adjacent commas), in which case the corresponding byte location will be filled with zeros.

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

### String constants

Single-character strings are stored in the most significant byte of a word, where the lower seven bits in that byte represent the ASCII value of the character, for example:

```
.WORD 'R'        ; = 0x52000000
```

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

```
.WORD  'ABCD'         ; = 0x44434241
```

### Related Information

`.SPACE` (Define Storage)

## 1.9.2. Assembler Controls

Controls start with a **$** as the first character on the line. Unknown controls are ignored after a warning is issued.

### Overview of assembler listing controls

| Control | Description |
|---|---|
| `$LIST ON/OFF` | Print / do not print source lines to list file |
| `$PAGE` | Generate form feed in list file |
| `$PAGE` *settings* | Define page layout for assembly list file |
| `$PRCTL` | Send control string to printer |
| `$STITLE` | Set program subtitle in header of assembly list file |
| `$TITLE` | Set program title in header of assembly list file |

### Overview of miscellaneous assembler controls

| Control | Description |
|---|---|
| `$CASE ON/OFF` | Case sensitive user names ON/OFF |
| `$DEBUG ON/OFF` | Generation of symbolic debug ON/OFF |
| `$IDENT LOCAL/GLOBAL` | Assembler treats labels by default as local or global |
| `$OBJECT` | Alternative name for the generated object file |
| `$WARNING OFF [`*num*`]` | Suppress all or some warnings |

## $CASE

### Syntax

```
$CASE  ON
$CASE  OFF
```

### Default

```
$CASE ON
```

### Description

With the `$CASE ON` and `$CASE OFF` controls you specify wether the assembler operates in case sensitive mode or not. By default the assembler operates in case sensitive mode. This means that all user-defined symbols and labels are treated case sensitive, so `LAB` and `Lab` are distinct.

Note that the instruction mnemonics, register names, directives and controls are always treated case insensitive.

### Example

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

### Related Information

Assembler option **--case-insensitive**

## $DEBUG

### Syntax

```
$DEBUG  ON
$DEBUG  OFF
```

### Default

```
$DEBUG OFF
```

### Description

With the `$DEBUG ON` and `$DEBUG OFF` controls you turn the generation of debug information on or off. (`$DEBUG ON` is similar to the assembler option **--debug-info=+asm,+local** (**-gal**).

### Example

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

### Related Information

Assembler option **--debug-info**

## $IDENT

### Syntax

```
$IDENT LOCAL
$IDENT GLOBAL
```

### Default

```
$IDENT LOCAL
```

### Description

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

By default the assembler treats all symbols as local symbols unless you have defined them to be global explicitly.

### Example

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

### Related Information

Assembler directive **.GLOBAL**

Assembler directive **.LOCAL**

Assembler option **--symbol-scope**

## $LIST ON/OFF

### Syntax

```
$LIST ON
$LIST OFF
```

### Default

```
$LIST ON
```

### Description

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

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

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

The listing still would not be disabled until another `$LIST OFF` control was issued.

### Example

```
   .SDECL ".mcstext.code",code
   .SECT  ".mcstext.code"
   ...  ; source line in list file
$LIST OFF
   ...  ; source line not in list file
$LIST ON
   ...  ; source line also in list file
```

### Related Information

Assembler option **--list-file**

Assembler function **@LST()**

## $OBJECT

### Syntax

```
$OBJECT "file"
$OBJECT OFF
```

### Default

```
$OBJECT
```

### 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.o"        ; generate object file x1.o
```

### Related Information

Assembler option **--output**

## $PAGE

### Syntax

**$PAGE** [*pagewidth*[,*pagelength*[,*blanktop*[,*blankbtm*[,*blankleft*]]]]

### Default

**$PAGE 132,72,0,0,0**

### Description

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

The arguments may be any positive absolute integer expression, and must be separated by commas.

| | |
|---|---|
| *pagewidth* | Number of columns per line. The default is 132, the minimum is 40. |
| *pagelength* | Total number of lines per page. The default is 72, the minimum is 10. As a special case, a page length of 0 turns off page breaks. |
| *blanktop* | Number of blank lines at the top of the page. The default is 0, the minimum is 0 and the maximum must be a value so that (*blanktop* + *blankbtm*) ≤ (*pagelength* - 10). |
| *blankbtm* | Number of blank lines at the bottom of the page. The default is 0, the minimum is 0 and the maximum must be a value so that (*blanktop* + *blankbtm*) ≤ (*pagelength* - 10). |
| *blankleft* | Number of blank columns at the left of the page. The default is 0, the minimum is 0, and the maximum must maintain the relationship: *blankleft* < *pagewidth*. |

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

### Example

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

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

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

### Related Information

Assembler option **--list-file**

## $PRCTL

### Syntax

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

### Description

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

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

You can specify the following arguments:

*expr* A byte expression which may be used to encode non-printing control characters, such as ESC.

*string* An assembler string, which may be of arbitrary length, up to the maximum assembler-defined limits.

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

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

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

### Example

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

### Related Information

Assembler option **--list-file**

## $STITLE

### Syntax

```
$STITLE "string"
```

### Default

```
$STITLE ""
```

### Description

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

The specified subtitle is valid until the assembler encounters a new `$STITLE` control. By default, the subtitle is empty.

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

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

### Example

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

### Related Information

Assembler option **--list-file**

Assembler control **$TITLE**

## $TITLE

### Syntax

```
$TITLE "string"
```

### Default

```
$TITLE ""
```

### Description

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

The specified title is valid until the assembler encounters a new `$TITLE` control. By default, the title is empty.

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

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

### Example

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

### Related Information

Assembler option **--list-file**

Assembler control **$STITLE**

## $WARNING OFF

### Syntax

**$WARNING OFF** [*number*]

### Default

All warnings are reported.

### Description

This control allows you to disable all or individual warnings. The *number* argument must be a valid warning message number.

### Example

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

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

### Related Information

Assembler option **--no-warnings**

# 1.10. Macro Operations

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

Some patterns contain variable entries which change for each repetition of the pattern. Others are subject to conditional assembly.

When a macro is called, the assembler executes the macro and replaces the call by the resulting in-line source statements. 'In-line' means that all replacements act as if they are on the same line as the macro call. The generated statements may contain substitutable arguments. The statements produced by a macro can be any processor instruction, almost any assembler directive, or any previously-defined macro. Source statements resulting from a macro call are subject to the same conditions and restrictions as any other statements.

Macros can be nested. The assembler processes nested macros when the outer macro is expanded.

## 1.10.1. Defining a Macro

The first step in using a macro is to define it.

The definition of a macro consists of three parts:

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

A macro definition takes the following form:

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

For more information on the definition see the description of the `.MACRO` directive.

The `.DUP`, `.DUPA`, `.DUPC`, and `.DUPF` directives are specialized macro forms to repeat a block of source statements. You can think of them as a simultaneous definition and call of an unnamed macro. The source statements between the `.DUP`, `.DUPA`, `.DUPC`, and `.DUPF` directives and the `.ENDM` directive follow the same rules as macro definitions.

## 1.10.2. Calling a Macro

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

```
[label] macro_name [argument[,argument]...]  [; comment]
```

where,

*label* An optional label that corresponds to the value of the location counter at the start of the macro expansion.

*macro_name* The name of the macro. This may not start in the first column.

*argument* One or more optional, substitutable arguments. Multiple arguments must be separated by commas.

*comment* An optional comment.

The following applies to macro arguments:

- Each argument must correspond one-to-one with the formal arguments of the macro definition. If the macro call does not contain the same number of arguments as the macro definition, the assembler issues a warning.
- If an argument has an embedded comma or space, you must surround the argument by single quotes (').
- You can declare a macro call argument as null in three ways:
  - enter delimiting commas in succession with no intervening spaces

    ```
    macroname ARG1,,ARG3 ; the second argument is a null argument
    ```

  - terminate the argument list with a comma, the arguments that normally would follow, are now considered null

    ```
    macroname ARG1,      ; the second and all following arguments are null
    ```

  - declare the argument as a null string
- No character is substituted in the generated statements that reference a null argument.

## 1.10.3. Using Operators for Macro Arguments

The assembler recognizes certain text operators within macro definitions which allow text substitution of arguments during macro expansion. You can use these operators for text concatenation, numeric conversion, and string handling.

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

### Example: Argument Concatenation Operator - \

Consider the following macro definition:

```
MAC_A .MACRO reg,val
   movl r\reg,val
   .ENDM
```

The macro is called as follows:

```
MAC_A 0,1
```

The macro expands as follows:

```
   movl r0,1
```

The macro preprocessor substitutes the character '0' for the argument `reg`, and the character '1' for the argument `val`. The concatenation operator (****) indicates to the macro preprocessor that the substitution characters for the arguments are to be concatenated with the character 'r'.

Without the '\' operator the macro would expand as:

```
   movl rreg,1
```

which results in an assembler error (invalid operand).

### Example: Decimal Value Operator - ?

Instead of substituting the formal arguments with the actual macro call arguments, you can also use the value of the macro call arguments.

Consider the following source code that calls the macro `MAC_A` after the argument `AVAL` has been set to `1`.

```
AVAL .SET  1
     MAC_A 0,AVAL
```

If you want to replace the argument `val` with the value of `AVAL` rather than with the literal string `'AVAL'`, you can use the **?** operator and modify the macro as follows:

```
MAC_A .MACRO reg,val
   movl r\reg,?val
   .ENDM
```

### Example: Hex Value Operator - %

The percent sign (**%**) is similar to the standard decimal value operator (**?**) except that it returns the hexadecimal value of a symbol.

Consider the following macro definition:

```
GEN_LAB    .MACRO  LAB,VAL,STMT
LAB\%VAL  STMT
     .ENDM
```

The macro is called after `NUM` has been set to 10:

```
NUM  .SET      10
     GEN_LAB   HEX,NUM,NOP
```

The macro expands as follows:

```
HEXA NOP
```

The `%VAL` argument is replaced by the character 'A' which represents the hexadecimal value 10 of the argument `VAL`.

## Example: Argument String Operator - "

To generate a literal string, enclosed by single quotes ('), you must use the argument string operator (") in the macro definition.

Consider the following macro definition:

```
STR_MAC    .MACRO  STRING
    .WORD  "STRING"
    .ENDM
```

The macro is called as follows:

```
    STR_MAC  ABCD
```

The macro expands as follows:

```
    .WORD   'ABCD'
```

Within double quotes `.DEFINE` directive definitions can be expanded. Take care when using constructions with single quotes and double quotes to avoid inappropriate expansions. Since `.DEFINE` expansion occurs before macro substitution, any `.DEFINE` symbols are replaced first within a macro argument string:

```
    .DEFINE LONG  'short'
STR_MAC    .MACRO  STRING
    .MESSAGE 'This is a LONG STRING'
    .MESSAGE "This is a LONG STRING"
    .ENDM
```

If the macro is called as follows:

```
    STR_MAC  sentence
```

it expands as:

```
    .MESSAGE 'This is a LONG STRING'
    .MESSAGE 'This is a short sentence'
```

## Macro Local Label Override Operator - ^

If you use labels in macros, the assembler normally generates another unique name for the labels (such as LAB__M_L000001).

The macro **^**-operator prevents name mangling on macro local labels.

Consider the following macro definition:

```
STA_Z .EQU    5

INIT  .MACRO  ARG, CNT
       MOV    R5,0x1
^LAB:
       .WORD  ARG
       ADD    R5,0x1
       ATUL   R5,CNT
       JBC    STA,STA_Z,^LAB
      .ENDM
```

The macro is called as follows:

```
       INIT 2,4
```

The macro expands as:

```
       MOV    R5,0x1
LAB:
       .WORD  2
       ADD    R5,0x1
       ATUL   R5,4
       JBC    STA,STA_Z,LAB
```

If you would have omitted the **^** operator, the macro preprocessor would choose another name for LAB because the label already exists. The macro would expand like:

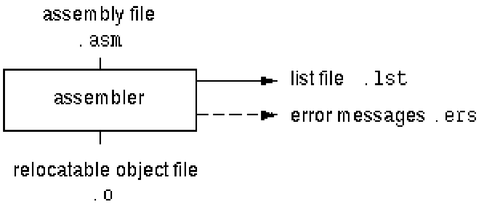
```
       MOV    R5,0x1
LAB__M_L000001:
       .WORD  2
       ADD    R5,0x1
       ATUL   R5,4
       JBC    STA,STA_Z,LAB__M_L000001
```

# Chapter 2. Using the Assembler

This chapter describes the assembly process and explains how to call the assembler.

The assembler converts hand-written assembly language programs into machine language, resulting in object files in the ELF/DWARF object format.

The assembler takes the following files for input and output:



The following information is described:

- The assembly process.
- How to call the assembler and how to use its options. An extensive list of all options and their descriptions is included in Section 5.1, *Assembler Options*.
- The various assembler optimizations.
- How to generate a list file.
- Types of assembler messages.

## 2.1. Assembly Process

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

### Phases of the assembly process

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

The assembler integrates file inclusion and macro facilities. See Section 1.10, *Macro Operations* for more information.

## 2.2. Calling the Assembler

The TASKING VX-toolset for MCS under Eclipse can use the internal builder (default) or the TASKING makefile generator (external builder) to build your entire project. After you have built your project, the output files are available in a subdirectory of your project directory, depending on the active configuration you have set in the **C/C++ Build » Settings** page of the **Project » Properties for** dialog.

### Building a project under Eclipse

You have several ways of building your project:

- Build Selected File(s) (101). This assembles the selected file(s) without calling the linker.

  1. In the C/C++ Projects view, select the files you want to assemble.

  2. Right-click in the C/C++ Projects view and select **Build Selected File(s)**.

- Build Individual Project (101).

  To build individual projects incrementally, select **Project » Build** *project*.

- Rebuild Project (101). This builds every file in the project whether or not a file has been modified since the last build. A rebuild is a clean followed by a build.

  1. Select **Project » Clean...**

  2. Enable the option **Start a build immediately** and click **OK**.

- Build Automatically. This performs a build of all projects whenever any project file is saved, such as your makefile.

  This way of building is not recommended, but to enable this feature select **Project » Build Automatically** and ensure there is a check mark beside the **Build Automatically** menu item. In order for this option to work, you must also enable option **Build on resource save (Auto build)** on the **Behaviour** tab of the **C/C++ Build** page of the **Project » Properties for** dialog.

### Select a target processor (core)

Processor options affect the invocation of all tools in the toolset. In Eclipse you only need to set them once.

1. From the **Project** menu, select **Properties for**

   *The Properties dialog appears.*

2. In the left pane, expand **C/C++ Build** and select **Processor**.

   *In the right pane the Processor page appears.*

3. From the **Processor selection** list, select a processor.

### To access the assembler options

1. From the **Project** menu, select **Properties for**

   *The Properties dialog appears.*

2. In the left pane, expand **C/C++ Build** and select **Settings**.

   *In the right pane the Settings appear.*

3. On the Tool Settings tab, select **Assembler**.

4. Select the sub-entries and set the options in the various pages.

You can find a detailed description of all assembler options in Section 5.1, *Assembler Options*.

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

```
asmcs [ [option]... [file]... ]...
```

The input file must be an assembly source file (`.asm` or `.mcs`).

## 2.3. How the Assembler Searches Include Files

When you use include files (with the `.INCLUDE` directive), you can specify their location in several ways. The assembler searches the specified locations in the following order:

1. If the `.INCLUDE` directive contains an absolute path name, the assembler looks for this file. If no path or a relative path is specified, the assembler looks in the same directory as the source file.

2. When the assembler did not find the include file, it looks in the directories that are specified in the **Assembler » Include Paths** page in the **C/C++ Build » Settings » Tool Settings** tab of the Project Properties dialog (equivalent to the **-I** command line option).

3. When the assembler did not find the include file (because it is not in the specified include directory or because no directory is specified), it looks in the path(s) specified in the environment variable `ASMCSINC`.

4. When the assembler still did not find the include file, it finally tries the default include directory relative to the installation directory.

### Example

Suppose that the assembly source file `test.asm` contains the following lines:

```
.INCLUDE 'myinc.inc'
```

You can call the assembler as follows:

```
asmcs -Imyinclude test.asm
```

First the assembler looks for the file `myinc.asm`, in the directory where `test.asm` is located. If the file is not there the assembler searches in the directory `myinclude`. If it was still not found, the assembler searches in the environment variable `ASMCSINC` and then in the default `include` directory.

## 2.4. Assembler Optimizations

The assembler can perform various optimizations that you can enable or disable.

1. From the **Project** menu, select **Properties for**

   *The Properties dialog appears.*

2. In the left pane, expand **C/C++ Build** and select **Settings**.

   *In the right pane the Settings appear.*

3. On the Tool Settings tab, select **Assembler » Optimization**.

4. Enable one or more optimizations.

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

### Optimize instruction size (option -Os/-OS)

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

## 2.5. Generating a List File

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

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

### To generate a list file

1. From the **Project** menu, select **Properties for**

   *The Properties dialog appears.*

2. In the left pane, expand **C/C++ Build** and select **Settings**.

   *In the right pane the Settings appear.*

3. On the Tool Settings tab, select **Assembler » List File**.

4. Enable the option **Generate list file**.

5. (Optional) Enable the options to include that information in the list file.

### Example on the command line (Windows Command Prompt)

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

```
asmcs -l test.asm
```

See Section 6.1, *Assembler List File Format*, for an explanation of the format of the list file.

## 2.6. Assembler Error Messages

The assembler reports the following types of error messages in the Problems view of Eclipse.

### F ( Fatal errors)

After a fatal error the assembler immediately aborts the assembly process.

### E (Errors)

Errors are reported, but the assembler continues assembling. No output files are produced unless you have set the assembler option **--keep-output-files** (the resulting output file may be incomplete).

### W (Warnings)

Warning messages do not result into an erroneous assembly output file. They are meant to draw your attention to assumptions of the assembler for a situation which may not be correct. You can control warnings in the **C/C++ Build » Settings » Tool Settings » Assembler » Diagnostics** page of the **Project » Properties for** menu (assembler option **--no-warnings**).

### Display detailed information on diagnostics

1. From the **Window** menu, select **Show View » Other » TASKING » Problems**.

   *The Problems view is added to the current perspective.*

2. In the Problems view right-click on a message.

   *A popup menu appears.*

3. Select **Detailed Diagnostics Info**.

   *A dialog box appears with additional information.*

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

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

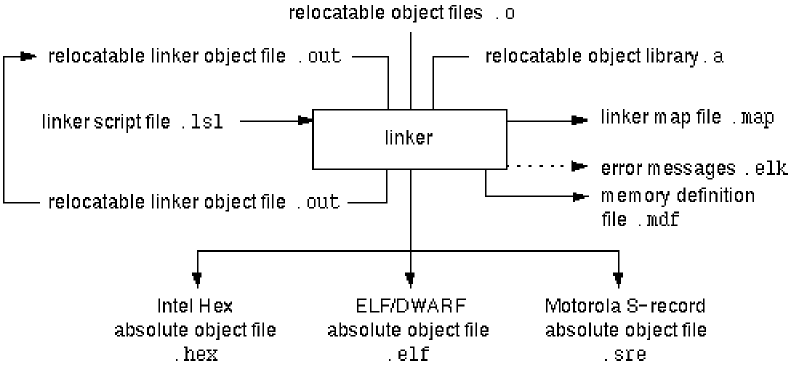
# Chapter 3. Using the Linker

This chapter describes the linking process, how to call the linker and how to control the linker with a script file.

The TASKING linker is a combined linker/locator. The linker phase combines relocatable object files (`.o` files, generated by the assembler), and libraries into a single relocatable linker object file (`.out`). The locator phase assigns absolute addresses to the linker object file and creates an absolute object file which you can load into a target processor. From this point the term linker is used for the combined linker/locator.

The linker can simultaneously link and locate all programs for all cores available on a target board. The target board may be of arbitrary complexity. A simple target board may contain one standard processor with some external memory that executes one task. A complex target board may contain multiple standard processors and DSPs combined with configurable IP-cores loaded in an FPGA. Each core may execute a different program, and external memory may be shared by multiple cores.

The linker takes the following files for input and output:


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

This chapter first describes the linking process. Then it describes how to call the linker and how to use its options. An extensive list of all options and their descriptions is included in Section 5.2, *Linker Options*.

To control the link process, you can write a script for the linker. This chapter shortly describes the purpose and basic principles of the Linker Script Language (LSL) on the basis of an example. A complete description of the LSL is included in Linker Script Language.

## 3.1. Linking Process

The linker combines and transforms relocatable object files (`.o`) into a single absolute object file. This process consists of two phases: the linking phase and the locating phase.

In the first phase the linker combines the supplied relocatable object files and libraries into a single relocatable object file. In the second phase, the linker assigns absolute addresses to the object file so it can actually be loaded into a target.

## Terms used in the linking process

| Term | Definition |
|---|---|
| Absolute object file | Object code in which addresses have fixed absolute values, ready to load into a target. |
| Address | A specification of a location in an address space. |
| Address space | The set of possible addresses. A core can support multiple spaces, for example in a Harvard architecture the addresses that identify the location of an instruction refer to *code* space, whereas addresses that identify the location of a data object refer to a *data* space. |
| Architecture | A description of the characteristics of a core that are of interest for the linker. This encompasses the address space(s) and the internal bus structure. Given this information the linker can convert logical addresses into physical addresses. |
| Copy table | A section created by the linker. This section contains data that specifies how the startup code initializes the data and BSS sections. For each section the copy table contains the following fields:<br>• action: defines whether a section is copied or zeroed<br>• destination: defines the section's address in RAM<br>• source: defines the sections address in ROM, zero for BSS sections<br>• length: defines the size of the section in MAUs of the destination space |
| Core | An instance of an architecture. |
| Derivative | The design of a processor. A description of one or more cores including internal memory and any number of buses. |
| Library | Collection of relocatable object files. Usually each object file in a library contains one symbol definition (for example, a function). |
| Logical address | An address as encoded in an instruction word, an address generated by a core (CPU). |
| LSL file | The set of linker script files that are passed to the linker. |
| MAU | Minimum Addressable Unit. For a given processor the number of bits between an address and the next address. This is not necessarily a byte or a word. |
| Object code | The binary machine language representation of the assembly source. |
| Physical address | An address generated by the memory system. |
| Processor | An instance of a derivative. Usually implemented as a (custom) chip, but can also be implemented in an FPGA, in which case the derivative can be designed by the developer. |
| Relocatable object file | Object code in which addresses are represented by symbols and thus relocatable. |
| Relocation | The process of assigning absolute addresses. |

| Term | Definition |
|---|---|
| Relocation information | Information about how the linker must modify the machine code instructions when it relocates addresses. |
| Section | A group of instructions and/or data objects that occupy a contiguous range of addresses. |
| Section attributes | Attributes that define how the section should be linked or located. |
| Target | The hardware board on which an application is executing. A board contains at least one processor. However, a complex target may contain multiple processors and external memory and may be shared between processors. |
| Unresolved reference | A reference to a symbol for which the linker did not find a definition yet. |

### 3.1.1. Phase 1: Linking

The linker takes one or more relocatable object files and/or libraries as input. A relocatable object file, as generated by the assembler, contains the following information:

- *Header information*: Overall information about the file, such as the code size, name of the source file it was assembled from, and creation date.

- *Object code*: Binary code and data, divided into various named sections. Sections are contiguous chunks of code that have to be placed in specific parts of the memory. The program addresses start at zero for each section in the object file.

- *Symbols*: Some symbols are exported - defined within the file for use in other files. Other symbols are imported - used in the file but not defined (external symbols). Generally these symbols are names of routines or names of data objects.

- *Relocation information*: A list of places with symbolic references that the linker has to replace with actual addresses. When in the code an external symbol (a symbol defined in another file or in a library) is referenced, the assembler does not know the symbol's size and address. Instead, the assembler generates a call to a preliminary relocatable address (usually 0000), while stating the symbol name.

- *Debug information*: Other information about the object code that is used by a debugger. The assembler optionally generates this information and can consist of line numbers, local symbols and descriptions of data structures.

The linker resolves the external references between the supplied relocatable object files and/or libraries and combines the files into a single relocatable linker object file.

The linker starts its task by scanning all specified relocatable object files and libraries. If the linker encounters an unresolved symbol, it remembers its name and continues scanning. The symbol may be defined elsewhere in the same file, or in one of the other files or libraries that you specified to the linker. If the symbol is defined in a library, the linker extracts the object file with the symbol definition from the library. This way the linker collects all definitions and references of all of the symbols.

Next, the linker combines sections with the same section name and attributes into single sections. The linker also substitutes (external) symbol references by (relocatable) numerical addresses where possible.

At the end of the linking phase, the linker either writes the results to a file (a single relocatable object file) or keeps the results in memory for further processing during the locating phase.

The resulting file of the linking phase is a single relocatable object file (.out). If this file contains unresolved references, you can link this file with other relocatable object files (.o) or libraries (.a) to resolve the remaining unresolved references.

With the linker command line option **--link-only**, you can tell the linker to only perform this linking phase and skip the locating phase. The linker complains if any unresolved references are left.

## 3.1.2. Phase 2: Locating

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

### Code modification

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

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

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

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

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

### Output formats

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

### Controlling the linker

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

- The memory installed in the embedded target system:

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

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

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

- The underlying hardware architecture of the target processor.

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

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

See also Section 3.6, *Controlling the Linker with a Script*.

## 3.2. Calling the Linker

In Eclipse you can set options specific for the linker. After you have built your project, the output files are available in a subdirectory of your project directory, depending on the active configuration you have set in the **C/C++ Build » Settings** page of the **Project » Properties for** dialog.

### Building a project under Eclipse

You have several ways of building your project:

- Build Individual Project (101).

  To build individual projects incrementally, select **Project » Build** *project*.

- Rebuild Project (101). This builds every file in the project whether or not a file has been modified since the last build. A rebuild is a clean followed by a build.

  1. Select **Project » Clean...**

  2. Enable the option **Start a build immediately** and click **OK**.

- Build Automatically. This performs a build of all projects whenever any project file is saved, such as your makefile.

  This way of building is not recommended, but to enable this feature select **Project » Build Automatically** and ensure there is a check mark beside the **Build Automatically** menu item. In order for this option to work, you must also enable option **Build on resource save (Auto build)** on the **Behaviour** tab of the **C/C++ Build** page of the **Project » Properties for** dialog.

### To access the linker options

1. From the **Project** menu, select **Properties for**

   *The Properties dialog appears.*

2. In the left pane, expand **C/C++ Build** and select **Settings**.

   *In the right pane the Settings appear.*

3. On the Tool Settings tab, select **Linker**.

4. Select the sub-entries and set the options in the various pages.

You can find a detailed description of all linker options in Section 5.2, *Linker Options*.

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

```
lmcs [ [option]... [file]... ]...
```

When you are linking multiple files, either relocatable object files (`.o`) or libraries (`.a`), it is important to specify the files in the right order.

Example:

```
lmcs -dtc27x.lsl test.o
```

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

## 3.3. Incremental Linking

With the TASKING linker it is possible to link incrementally. Incremental linking means that you link some, but not all `.o` modules to a relocatable object file `.out`. In this case the linker does not perform the locating phase. With the second invocation, you specify both new `.o` files as the `.out` file you had created with the first invocation.

> Incremental linking is only possible on the command line.

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

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

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

## 3.4. Importing Binary Files

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

For example:

```
.extern  __lc_ub_my_mp3; /* linker labels */
.extern  __lc_ue_my_mp3;
```

If you want to use the export functionality of Eclipse, the binary file has to be part of your project.

## 3.5. Linker Optimizations

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

### To enable or disable optimizations

1. From the **Project** menu, select **Properties for**

   *The Properties dialog appears.*

2. In the left pane, expand **C/C++ Build** and select **Settings**.

   *In the right pane the Settings appear.*

3. On the Tool Settings tab, select **Linker » Optimization**.

4. Enable one or more optimizations.

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

### Delete unreferenced sections (option -Oc/-OC)

This optimization removes unused sections from the resulting object file.

### First fit decreasing (option -Ol/-OL)

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

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

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

### Compress copy table (option -Ot/-OT)

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

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

### Delete duplicate code (option -Ox/-OX)

### Delete duplicate constant data (option -Oy/-OY)

These two optimizations remove code and constant data that is defined more than once, from the resulting object file.

## 3.6. Controlling the Linker with a Script

With the options on the command line you can control the linker's behavior to a certain degree. From Eclipse it is also possible to determine where your sections will be located, how much memory is available, which sorts of memory are available, and so on. Eclipse passes these locating directions to the linker via a script file. If you want even more control over the locating process you can supply your own script.

The language for the script is called the *Linker Script Language*, or shortly LSL. You can specify the script file to the linker, which reads it and locates your application exactly as defined in the script. If you do not specify your own script file, the linker always reads a standard script file which is supplied with the toolset.

## 3.6.1. Purpose of the Linker Script Language

The Linker Script Language (LSL) serves three purposes:

1. It provides the linker with a definition of the target's core architecture. This definition is supplied with the toolset.
2. It provides the linker with a specification of the memory attached to the target processor.
3. It provides the linker with information on how your application should be located in memory. This gives you, for example, the possibility to create overlaying sections.

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

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

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

The complete LSL syntax is described in Chapter 7, *Linker Script Language (LSL)*.

## 3.6.2. Eclipse and LSL

In Eclipse you can specify the size of the stack and heap; the physical memory attached to the processor; identify that particular address ranges are reserved; and specify which sections are located where in memory. Eclipse translates your input into an LSL file that is stored in the project directory under the name *project_name*`.lsl` and passes this file to the linker. If you want to learn more about LSL you can inspect the generated file *project_name*`.lsl`.

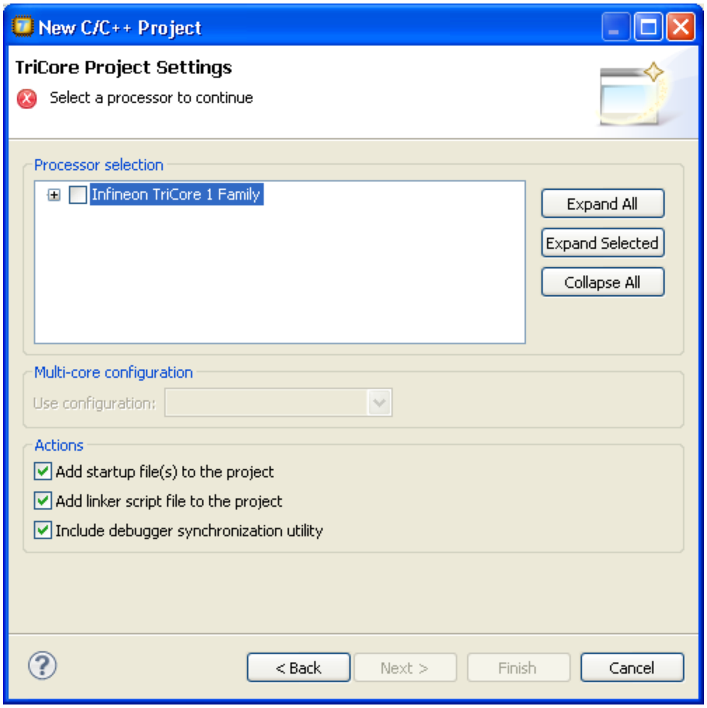
Because an MCS project is part of a TriCore project you only need to specify an LSL file to the TriCore project.

### To add a generated Linker Script File to your project

1. From the **File** menu, select **File » New » TASKING TriCore C/C++ Project**.

   *The New C/C++ Project wizard appears.*

2. Fill in the project settings in each dialog and click **Next >** until the following dialog appears.


New C/C++ Project
TriCore Project Settings
Select a processor to continue
Processor selection
Infineon TriCore 1 Family
Expand All
Expand Selected
Collapse All
Multi-core configuration
Use configuration:
Actions
☑ Add startup file(s) to the project
☑ Add linker script file to the project
☑ Include debugger synchronization utility
< Back
Next >
Finish
Cancel

3. Enable the option **Add linker script file to the project** and click **Finish**.

   *Eclipse creates your project and the file "project_name*`.lsl`*" in the project directory.*

If you do not add the linker script file here, you can always add it later with **File » New » Linker Script File (LSL)**

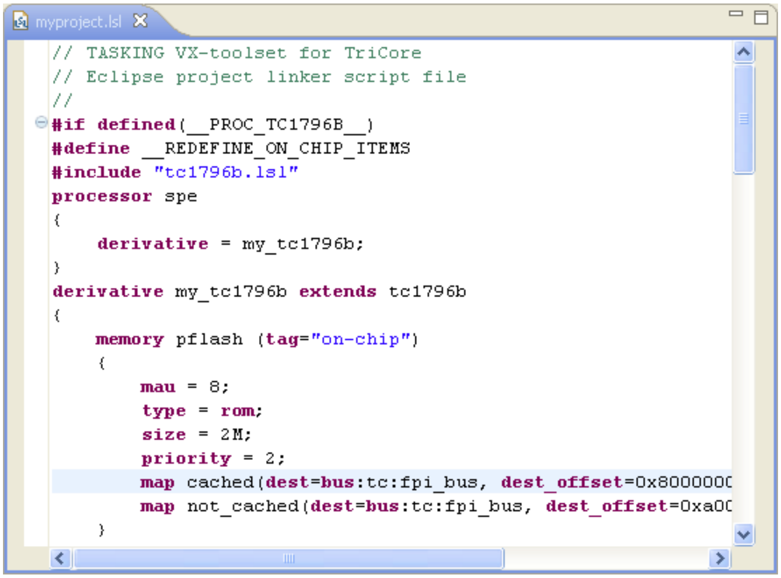
## To change the Linker Script File in Eclipse

There are two ways of changing the LSL file in Eclipse.

- You can change the LSL file directly in an editor.

  1. Double-click on the file *project_name*`.lsl`.

     *The project LSL file opens in the editor area.*



```
// TASKING VX-toolset for TriCore
// Eclipse project linker script file
//
#if defined(__PROC_TC1796B__)
#define __REDEFINE_ON_CHIP_ITEMS
#include "tc1796b.lsl"
processor spe
{
    derivative = my_tc1796b;
}
derivative my_tc1796b extends tc1796b
{
    memory pflash (tag="on-chip")
    {
        mau = 8;
        type = rom;
        size = 2M;
        priority = 2;
        map cached(dest=bus:tc:fpi_bus, dest_offset=0x8000000
        map not_cached(dest=bus:tc:fpi_bus, dest_offset=0xa0
    }
```

2. You can edit the LSL file directly in the *project_name*`.lsl` editor.

   *A * appears in front of the name of the LSL file to indicate that the file has changes.*

3. Click 💾 or select **File » Save** to save the changes.

- You can also make changes to the property pages Memory and Stack/Heap.

1. From the **Project** menu, select **Properties for**

   *The Properties dialog appears.*

2. In the left pane, expand **C/C++ Build** and select **Memory** or **Stack/Heap**.

   *In the right pane the corresponding property page appears.*

3. Make changes to memory and/or stack/heap and click **OK**.

   *The project LSL file is updated automatically according to the changes you make in the pages.*

You can quickly navigate through the LSL file by using the Outline view (**Window » Show View » Outline**).

## 3.6.3. Structure of a Linker Script File

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

## The architecture definition (required)

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

This specification is normally written by Altium. Altium supplies LSL files in the `include.lsl` directory. The file `tc_arch.lsl` defines the base architecture for all cores and includes an interrupt vector table (`inttab.lsl`) and an trap vector table (`traptab.lsl`). The files `tc1v1_3.lsl`, `tc1v1_3_1.lsl` and `tc1v1_6.lsl` extend the base architecture for each TriCore core.

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

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

## The derivative definition

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

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

Altium supplies LSL files for each derivative (`*derivative*.lsl`). When you build an ASIC or use a derivative that is not (yet) supported by the TASKING tools, you may have to write a derivative definition.

## The processor definition

The *processor definition* describes an instance of a derivative. A processor definition is only needed in a multi-processor embedded system. It allows you to define multiple processors of the same type.

If for a derivative 'A' no processor is defined in the LSL file, the linker automatically creates a processor named 'A' of derivative 'A'. This is why for single-processor applications it is enough to specify the derivative in the LSL file.

## The memory and bus definitions (optional)

Memory and bus definitions are used within the context of a derivative definition to specify internal memory and on-chip buses. In the context of a board specification the memory and bus definitions are used to define external (off-chip) memory and buses. Given the above definitions the linker can convert a logical address into an offset into an on-chip or off-chip memory device.

## The board specification

The processor definition and memory and bus definitions together form a board specification. LSL provides language constructs to easily describe single-core and heterogeneous or homogeneous multi-core systems. The board specification describes all characteristics of your target board's system buses, memory devices, I/O sub-systems, and cores that are of interest to the linker. Based on the information provided in the board specification the linker can for each core:

- convert a logical address to an offset within a memory device
- locate sections in physical memory
- maintain an overall view of the used and free physical memory within the whole system while locating

## The section layout definition (optional)

The optional section layout definition enables you to exactly control where input sections are located. Features are provided such as: the ability to place sections at a given address, to place sections in a given order, and to overlay code and/or data sections.

## Example: Skeleton of a Linker Script File

A linker script file that defines a derivative "X'" based on the TC1V1.6.X and MCS architecture, its external memory and how sections are located in memory, may have the following skeleton:

```
architecture TC1V1.6.X
{
    // Specification of the TC1V1.6.X core architecture.
    // Written by Altium.
}

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

derivative X  // derivative name is arbitrary
{
    // Specification of the derivative.
    // Written by Altium.
    core tc0         // always specify the core(s)
    {
       architecture = TC1V1.6.X;
       // ...
    }

    core tc1         // always specify the core(s)
    {
       architecture = TC1V1.6.X;
       // ...
    }

    core tc2         // always specify the core(s)
    {
       architecture = TC1V1.6.X;
       // ...
    }

    core mcs00       // always specify the core(s)
    {
       architecture = MCS;
       // ...
    }

    core mcs01       // always specify the core(s)
    {
       architecture = MCS;
       // ...
    }

    core vtc         // virtual core
    {
       architecture = TC1V1.6.X;
       import tc0;
       import tc1;
       import tc2;
    }

    bus sri          // internal bus
    {
       // maps to bus "fpi_bus" in real "tc0", "tc1", ... cores
       // and virtual core "vtc"
    }

    // internal memory
}

processor mpe        // multi-core processor name
{
    derivative = X;
}

memory ext_name
{
    // external memory definition
}

section_layout mpe:vtc:linear    // section layout
{
    // section placement statements
    // sections are located in address space 'linear'
    // of virtual core 'vtc' of processor 'mpe'
}
```

See for example the file `tc27x.lsl` in the directory `include.lsl` for an actual implementation.

## Overview of LSL files delivered by Altium

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

| LSL file | Description |
|---|---|
| `tc_arch.lsl` | Defines the base architecture (TC) for all generic TriCore cores. It includes the files `inttab.lsl` and `traptab.lsl`. |
| `tc_mc_arch.lsl` | Defines the base architecture (TC) for all multi-core TriCore cores. |
| `mcs_arch.lsl` | Defines the base architecture (MCS) for all MCS cores. |
| `inttab.lsl` | Defines the interrupt vector table. It is included in the file `tc_arch.lsl`. |
| `inttab{0\|1\|2}.lsl` | Defines a core specific interrupt vector table. It is included in derivative LSL files that have multi-core support. |
| `traptab.lsl` | Defines the trap vector table. It is included in the file `tc_arch.lsl`. |
| `traptab{0\|1\|2}.lsl` | Defines a core specific trap vector table. It is included in derivative LSL files that have multi-core support. |
| `tc1v1_3.lsl`<br>`tc1v1_3_1.lsl`<br>`tc1v1_6.lsl`<br>`tc1v1_6_x.lsl` | Extends the base architecture for cores TC1V1.3, TC1V1.3.1, TC1V1.6 or TC1V1.6.X. It includes the file `tc_arch.lsl` or `tc_mc_arch.lsl` and `mcs_arch.lsl`. |
| *`derivative`*`.lsl` | Defines the derivative and defines a single processor. Contains a memory definition and section layout. It includes one of the files `tc`*`version`*`.lsl`. The selection of the derivative is based on your CPU selection (control program option **--cpu**). |
| `userdef13.lsl`<br>`userdef131.lsl`<br>`userdef16.lsl`<br>`userdef16x.lsl` | Defines a user defined derivative for cores TC1V1.3, TC1V1.3.1, TC1V1.6 or TC1V1.6.X and defines a single processor for TC1V1.3, TC1V1.3.1 and TC1V1.6 and a multi-core processor for TC1V1.6.X. |
| `template.lsl` | This file is used by Eclipse as a template for the project LSL file. It includes the file *`derivative`*`.lsl` based on your CPU selection. The CPU is specified by the __CPU__ macro. |
| `default.lsl` | Contains a default memory definition and section layout based on the tc1796b derivative. This file is used on a command line invocation of the tools, when no CPU is selected (no option **--cpu**). It includes the file `extmem.lsl`. |
| `extmem.lsl` | Template file with a specification of the external memory attached to the target processor. |

When you select to add a linker script file when you create a project in Eclipse, Eclipse makes a copy of the file `template.lsl` and names it "*project_name*`.lsl`". On the command line, the linker uses the file `default.lsl`, unless you specify another file with the linker option **--lsl-file** (**-d**).

## 3.6.4. The Architecture Definition

Although you will probably not need to write an architecture definition (unless you are building your own processor core) it helps to understand the Linker Script Language and how the definitions are interrelated.

Within an *architecture definition* the characteristics of a target processor core that are important for the linking process are defined. These include:

- space definitions: the logical address spaces and their properties
- bus definitions: the I/O buses of the core architecture
- mappings: the address translations between logical address spaces, the connections between logical address spaces and buses and the address translations between buses

### Address spaces

A logical address space is a memory range for which the core has a separate way to encode an address into instructions. Most microcontrollers and DSPs support multiple address spaces. For example, the TriCore's 32-bit linear address space encloses 16 24-bit sub-spaces and 16 14-bit sub-spaces. Normally, the size of an address space is $2^N$, with *N* the number of bits used to encode the addresses.

The relation of an address space with another address space can be one of the following:

- one space is a subset of the other. These are often used for "small" absolute or relative addressing.
- the addresses in the two address spaces represent different locations so they do not overlap. This means the core must have separate sets of address lines for the address spaces. For example, in Harvard architectures we can identify at least a code and a data memory space.

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

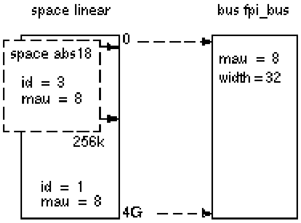
| Space | Id | MAU | Description | ELF sections |
|---|---|---|---|---|
| linear | 1 | 1 | Linear address space. | .text, .bss, .data, .rodata, table, istack, ustack |
| abs24 | 2 | 8 | Absolute 24-bit addressable space | |
| abs18 | 3 | 8 | Absolute 18-bit addressable space. | .zdata, .zbss |
| csa | 4 | 8 | Context Save Area | csa.* |

The MCS, which is part of TriCore v1.6.x derivatives, such as TC27X, has one address space for architecture `MCS` as defined in `mcs_arch.lsl`

| Space | Id | MAU | Description | ELF sections |
|---|---|---|---|---|
| mcs | 1 | 8 | MCS address space | .mcstext, .mcsdata |

## The TriCore architecture in LSL notation

The best way to write the architecture definition, is to start with a drawing. The following figure shows a part of the TriCore architecture `TC` as defined in `tc_mc_arch.lsl`:



The figure shows two address spaces called `linear` and `abs18`. The address space `abs18` is a subset of the address space `linear`. All address spaces have attributes like a number that identifies the logical space (id), a MAU and an alignment. In LSL notation the definition of these address spaces looks as follows:

```
space linear
{
     id = 1;
     mau = 8;

     map (src_offset=0x00000000, dest_offset=0x00000000,
          size=4G, dest=bus:fpi_bus);
}

space abs18
{
     id = 3;
     mau = 8;

     map (src_offset=0x00000000, dest_offset=0x00000000,
          size=16k, dest=space:linear);
     map (src_offset=0x10000000, dest_offset=0x10000000,
          size=16k, dest=space:linear);
     map (src_offset=0x20000000, dest_offset=0x20000000,
          size=16k, dest=space:linear);
   //...
}
```

The keyword `map` corresponds with the arrows in the drawing. You can map:

- address space => address space
- address space => bus
- memory => bus (not shown in the drawing)
- bus => bus (not shown in the drawing)

Next the internal bus, named `fpi_bus` must be defined in LSL:

```
bus fpi_bus
{
     mau = 8;
     width = 32;  // there are 32 data lines on the bus
}
```

This completes the LSL code in the architecture definition. Note that all code above goes into the architecture definition, thus between:

```
architecture TC1V1.6.X
{
    // All code above goes here.
}
```

## 3.6.5. The Derivative Definition

Although you will probably not need to write a derivative definition (unless you are using multiple cores that both access the same memory device) it helps to understand the Linker Script Language and how the definitions are interrelated.

A *derivative* is the design of a processor, as implemented on a chip (or FPGA). It comprises one or more cores and on-chip memory. The derivative definition includes:

- core definition: an instance of a core architecture
- bus definition: the I/O buses of the core architecture
- memory definitions: internal (or on-chip) memory

### Core

Each derivative must have at least one core and each core must have a specification of its core architecture. This core architecture must be defined somewhere in the LSL file(s).

```
core mcs00
{
    architecture = MCS;
    copytable_space = vtc:linear; // use copytable from core vtc
}
```

In a multi-core environment you can combine multiple cores with the same architecture into a single link task. This is done by importing one or more cores into a root core. The imported cores share a single symbol namespace. The address spaces in each imported core must have a unique ID in the link task. For each imported core is specified that the space IDs of the imported core start at a specific offset. If writable sections for a core must be initialized by using the copy table of a different core, this is specified by a `copytable_space`. The following example is part of `tc27x.lsl` delivered with the product.

```
core tc0 // core 0
{
    architecture = TC1V1.6.X;
    space_id_offset = 100; // add 100 to all space IDs in
                           // the architecture definition
    copytable_space = vtc:linear; // use copytable from core vtc
}
core tc1 // core 1
{
    architecture = TC1V1.6.X;
    space_id_offset = 200; // add 200 to all space IDs in
                           // the architecture definition
    copytable_space = vtc:linear; // use copytable from core vtc
}

core tc2 // core 2
{
    architecture = TC1V1.6.X;
    space_id_offset = 300; // add 300 to all space IDs in
                           // the architecture definition
    copytable_space = vtc:linear; // use copytable from core vtc
}

core vtc
{
    architecture = TC1V1.6.X;
    import tc0; // add all address spaces of tc0 for linking
    import tc1; // add all address spaces of tc1 for linking
    import tc2; // add all address spaces of tc2 for linking
}
```

### Bus

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

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

### Memory

External memory is usually described in a separate memory definition, but you can specify on-chip memory for a derivative. For example:

```
memory dspr0
{
     mau = 8;
     size = 128k;
     type = ram;
     map (dest=bus:tc0:fpi_bus, dest_offset=0xd0000000,
          size=128k, priority=8);
     map (dest=bus:sri, dest_offset=0xd8000000, size=128k);
}
```

This completes the LSL code in the derivative definition. Note that all code above goes into the derivative definition, thus between:

```
derivative X    // name of derivative
{
    // All code above goes here
}
```

## 3.6.6. The Processor Definition

The processor definition is only needed when you write an LSL file for a multi-processor embedded system. The processor definition explicitly instantiates a derivative, allowing multiple processors of the same type.

```
processor name
{
    derivative = derivative_name;
}
```

If no processor definition is available that instantiates a derivative, a processor is created with the same name as the derivative.

Altium defines a “multi processor environment” (mpe) in each *derivative*.lsl file. For example:

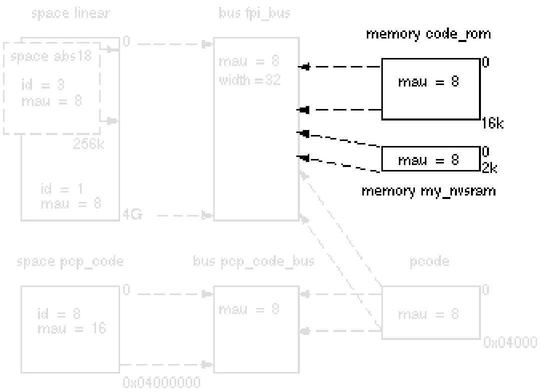
```
processor mpe
{
    derivative = tc27x;
}
```

## 3.6.7. The Memory Definition

Once the core architecture is defined in LSL, you may want to extend the processor with external (or off-chip) memory. You need to specify the location and size of the physical external memory devices in the target system.

The principle is the same as defining the core's architecture but now you need to fill the memory definition:

```
memory name
{
    // memory definitions
}
```



Suppose your embedded system has 16 kB of external ROM, named `code_rom` and 2 kB of external NVRAM, named `my_nvsram`. Both memories are connected to the bus `fpi_bus`. In LSL this looks like:

```
memory code_rom
{
    mau = 8;
    size = 16k;
    type = rom;
    map( dest=bus:mpe:fpi_bus, dest_offset=0xa0000000, size=16k );
}

memory my_nvsram
{
    mau  = 8;
    size = 2k;
    type = nvram;
    map( dest=bus:mpe:fpi_bus, dest_offset=0xc0000000, size=2k );
}
```

If you use a different memory layout than described in the LSL file supplied for the target core, you can specify this in Eclipse or you can specify this in a separate LSL file and pass both the LSL file that describes the core architecture and your LSL file that contains the memory specification to the linker.

## To add memory using Eclipse

1. From the **Project** menu, select **Properties for**

   *The Properties dialog appears.*

2. In the left pane, expand **C/C++ Build** and select **Memory**.

   *In the right pane the Memory page appears.*

3. Open the **Memory** tab and click on the **Add...** button.

   *The Add new memory dialog appears.*

4. Enter the memory name (for example `my_nvsram`), type (for example `nvram`) and size.

5. Click on the **Add...** button.

   *The Add new mapping dialog appears.*

6. You have to specify at least one mapping. Enter the mapping name (optional), address, size and destination and click **OK**.

   *The new mapping is added to the list of mappings.*

7. Click **OK**.

   *The new memory is added to the list of memories (user memory).*

8. Click **OK** to close the Properties dialog.

   *The updated settings are stored in the project LSL file.*

If you make changes to the on-chip memory as defined in the architecture LSL file, the memory is copied to your project LSL file and the line `#define __REDEFINE_ON_CHIP_ITEMS` is added. If you remove all the on-chip memory from your project LSL file, also make sure you remove this define.

## 3.6.8. The Section Layout Definition: Locating Sections

Once you have defined the internal core architecture and optional memory, you can actually define where your application must be located in the physical memory.

Sections have a name, an indication (section type) in which address space it should be located and attributes like writable or read-only.

In the section layout definition you can exactly define how input sections are placed in address spaces, relative to each other, and what their absolute run-time and load-time addresses will be.

### Section placement

Suppose we want to save sections in non-volatile (battery back-upped) memory. This is the memory `my_nvsram` from the example in Section 3.6.7, *The Memory Definition*.

To control the locating of sections, you need to write one or more section definitions in the LSL file. At least one for each address space where you want to change the default behavior of the linker. In our example, we need to locate sections in the address space `data`:

```
section_layout :vtc:linear
{
    // Section placement statements
}
```

To locate sections, you must create a group in which you select sections from your program. For our example, we need to define one group, which contains the section `.mcsdata.non_volatile`. All other sections are located using the defaults specified in the architecture definition. Section

`.mcsdata.non_volatile` should be placed in non-volatile ram. To achieve this, the run address refers to our non-volatile memory called `my_nvsram`.

```
group ( ordered, run_addr = mem:my_nvsram )
{
     select ".mcsdata.non_volatile";
}
```

This completes the LSL file for the sample architecture and sample program. You can now invoke the linker with this file and the sample program to obtain an application that works for this architecture.

For a complete description of the Linker Script Language, refer to Chapter 7, *Linker Script Language (LSL)*.

### 3.6.8.1. Locating Code and Data Sections in Separate Pages

When code and data are in the same memory page, this will have a negative effect on the run-time speed. To have the best performance, code must end up in `mp0` and data in `mp1`.

The following example shows how to do this in LSL. Note that for "code in mp0, data in mp0", the directions need to be reversed. Only a `section_layout` with `high_to_low` is needed since `low_to_high` is the default.

```
section_layout mpe:mcs00:mcs (direction=high_to_low)
{
    group (contiguous)
    {
        select ".mcstext";
        select ".mcstext.*";
    }
}

section_layout mpe:mcs00:mcs (direction=low_to_high)
{
    group (contiguous)
    {
        select ".mcsdata";
        select ".mcsdata.*";
    }
}
```

## 3.7. Linker Labels

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

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

| Label | Description |
|---|---|
| `_lc_ub_`*`name`*<br>`_lc_b_`*`name`* | Begin of section *name*. Also used to mark the begin of the stack or heap or copy table. |
| `_lc_ue_`*`name`*<br>`_lc_e_`*`name`* | End of section *name*. Also used to mark the end of the stack or heap. |
| `_lc_cb_`*`name`* | Start address of an overlay section in ROM. |
| `_lc_ce_`*`name`* | End address of an overlay section in ROM. |
| `_lc_gb_`*`name`* | Begin of group *name*. This label appears in the output file even if no reference to the label exists in the input file. |
| `_lc_ge_`*`name`* | End of group *name*. This label appears in the output file even if no reference to the label exists in the input file. |
| `_lc_s_`*`name`* | Variable *name* is mapped through memory in shared memory situations. |

The linker only allocates space for the stack and/or heap when a reference to either of the section labels exists in one of the input object files.

Additionally, the linker script file defines the following symbols:

| Symbol | Description |
|---|---|
| `_lc_cp` | Start of copy table. Same as `_lc_ub_table`. The copy table gives the source and destination addresses of sections to be copied. This table will be generated by the linker only if this label is used. |
| `_lc_bh` | Begin of heap. Same as `_lc_ub_heap`. |
| `_lc_eh` | End of heap. Same as `_lc_ue_heap`. |

If there is no LSL file in your project, select **File » New » Linker Script File (LSL)**, add the lines that define the symbol. Add the LSL file to the linker options (**Tool Options » Linker » Script File » Linker script file (.lsl)**).

When the MCS linked project (`.out`) is linked with a TriCore project, then the TriCore LSL file also needs this addition.

## 3.8. Generating a Map File

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

### To generate a map file

1. From the **Project** menu, select **Properties for**

   *The Properties dialog appears.*

2. In the left pane, expand **C/C++ Build** and select **Settings**.

   *In the right pane the Settings appear.*

3. On the Tool Settings tab, select **Linker » Map File**.

4. Enable the option **Generate XML map file format (.mapxml) for map file viewer**.

5. (Optional) Enable the option **Generate map file (.map)**.

6. (Optional) Enable the options to include that information in the map file.

### Example on the command line (Windows Command Prompt)

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

```
lmcs --map-file test.o
```

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

See Section 6.2, *Linker Map File Format*, for an explanation of the format of the map file.

## 3.9. Linker Error Messages

The linker reports the following types of error messages in the Problems view of Eclipse.

### F ( Fatal errors)

After a fatal error the linker immediately aborts the link/locate process.

### E (Errors)

Errors are reported, but the linker continues linking and locating. No output files are produced unless you have set the linker option**--keep-output-files**.

### W (Warnings)

Warning messages do not result into an erroneous output file. They are meant to draw your attention to assumptions of the linker for a situation which may not be correct. You can control warnings in the **C/C++ Build » Settings » Tool Settings » Linker » Diagnostics** page of the **Project » Properties for** menu (linker option **--no-warnings**).

### I (Information)

Verbose information messages do not indicate an error but tell something about a process or the state of the linker. To see verbose information, use the linker option**--verbose**.

## S (System errors)

System errors occur when internal consistency checks fail and should never occur. When you still receive the system error message

```
S6##: message
```

please report the error number and as many details as possible about the context in which the error occurred.

## Display detailed information on diagnostics

1. From the **Window** menu, select **Show View » Other » TASKING » Problems**.

   *The Problems view is added to the current perspective.*

2. In the Problems view right-click on a message.

   *A popup menu appears.*

3. Select **Detailed Diagnostics Info**.

   *A dialog box appears with additional information.*

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

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

# Chapter 4. Using the Utilities

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

**ccmcs** A control program. The control program invokes all tools in the toolset and lets you quickly generate an absolute object file from assembly source input files. Eclipse uses the control program to call the assembler and linker.

**mkmcs** A utility program to maintain, update, and reconstruct groups of programs. The make utility looks whether files are out of date, rebuilds them and determines which other files as a consequence also need to be rebuilt.

**amk** The make utility which is used in Eclipse. It supports parallelism which utilizes the multiple cores found on modern host hardware.

**armcs** An archiver. With this utility you create and maintain library files with relocatable object modules (`.o`) generated by the assembler.

**cnvba2ta.pl** A Perl script to convert Bosch MCS assembly to TASKING assembly for MCS.

## 4.1. Control Program

The control program is a tool that invokes all tools in the toolset for you. It provides a quick and easy way to generate the final absolute object file out of your sources without the need to invoke the assembler and linker manually.

Eclipse uses the control program to call the assembler and linker, but you can call the control program from the command line. The invocation syntax is:

```
ccmcs [ [option]... [file]... ]...
```

### Recognized input files

- Files with a `.asm` or `.mcs` suffix are interpreted as hand-written assembly source files which have to be passed to the assembler.

- Files with a `.a` suffix are interpreted as library files and are passed to the linker.

- Files with a `.o` suffix are interpreted as object files and are passed to the linker.

- Files with a `.out` suffix are interpreted as linked object files and are passed to the locating phase of the linker. The linker accepts only one `.out` file in the invocation.

- Files with a `.lsl` suffix are interpreted as linker script files and are passed to the linker.

### Options

The control program accepts several command line options. If you specify an unknown option to the control program, the control program looks if it is an option for a specific tool. If so, it passes the option directly to the tool. However, it is recommended to use the control program options **--pass-*** (**-Wa**, **-Wl**) to pass arguments directly to tools.

For a complete list and description of all control program options, see Section 5.3, *Control Program Options*.

### Example with verbose output

```
ccmcs --verbose --cpu=tc27x test.asm
```

The control program calls all tools in the toolset and generates the absolute object file `test.elf`. With option **--verbose** (**-v**) you can see how the control program calls the tools:

```
+ "path\asmcs" -Ctc27x -o cc3248b.o test.asm
+ "path\lmcs" -o test.elf -dtc27x.lsl
     --map-file cc3248b.o
```

The control program produces unique filenames for intermediate steps in the build process (such as `cc3248b.o` in the example above) which are removed afterwards, unless you specify command line option **--keep-temporary-files** (**-t**).

### Example with argument passing to a tool

```
ccmcs --pass-assembler=-ga test.asm
```

The option **-ga** is directly passed to the assembler.

## 4.2. Make Utility mkmcs

If you are working with large quantities of files, or if you need to build several targets, it is rather time-consuming to call the individual tools to assemble, link and locate all your files.

You save already a lot of typing if you use the control program and define an options file. You can even create a batch file or script that invokes the control program for each target you want to create. But with these methods all files are completely assembled and linked to obtain the target file, even if you changed just one assembly source. This may demand a lot of (CPU) time on your host.

The make utility **mkmcs** is a tool to maintain, update, and reconstruct groups of programs. The make utility looks which files are out-of-date and only recreates these files to obtain the updated target.

### Make process

In order to build a target, the make utility needs the following input:

- the target it should build, specified as argument on the command line
- the rules to build the target, stored in a file usually called `makefile`

> In addition, the make utility also reads the file `mkmcs.mk` which contains predefined rules and macros. See Section 4.2.2, *Writing a Makefile*.

The `makefile` contains the relationships among your files (called *dependencies*) and the commands that are necessary to create each of the files (called *rules*). Typically, the absolute object file (`.elf`) is updated when one of its dependencies has changed. The absolute file depends on `.o` files and libraries that must be linked together. The `.o` files on their turn depend on `.asm` files that must be assembled. In the `makefile` this looks like:

```
test.o   : test.asm                 # dependency
           asmcs test.asm           # rule

test.elf : test.o
           lmcs test.o -o test.elf --map-file
```

You can use any command that is valid on the command line as a rule in the `makefile`. So, rules are not restricted to invocation of the toolset.

## Example

To build the target `test.elf`, call **mkmcs** with one of the following lines:

```
mkmcs test.elf
```

```
mkmcs -fmymake.mak test.elf
```

By default the make utility reads the file `makefile` so you do not need to specify it on the command line. If you want to use another name for the makefile, use the option **-f**.

If you do not specify a target, **mkmcs** uses the first target defined in the makefile. In this example it would build `test.src` instead of `test.elf`.

Based on the sample invocation, the make utility now tries to build `test.elf` based on the makefile and performs the following steps:

1. From the makefile the make utility reads that `test.elf` depends on `test.o`.
2. If `test.o` does not exist or is out-of-date, the make utility first tries to build this file and reads from the makefile that `test.o` depends on `test.asm`.
3. The make utility creates `test.o` by executing the rule for it: `asmcs test.asm`.
4. There are no other files necessary to create `test.elf` so the make utility now can use `test.o` to create `test.elf` by executing the rule: `lmcs test.o -o test.elf ...`

The make utility has now built `test.elf` but it only used the assembler to update `test.o` and the linker to create `test.elf`.

If you compare this to the control program:

```
ccmcs test.asm
```

This invocation has the same effect but now *all files* are reassembled, linked and located.

## 4.2.1. Calling the Make Utility

You can only call the make utility from the command line. The invocation syntax is:

```
mkmcs [ [option]... [target]... [macro=def]... ]
```

For example:

```
mkmcs test.elf
```

*target* You can specify any target that is defined in the makefile. A target can also be one of the intermediate files specified in the makefile.

*macro***=***def* Macro definition. This definition remains fixed for the **mkmcs** invocation. It overrides any regular definitions for the specified macro within the makefiles and from the environment. It is inherited by subordinate **mkmcs**'s but act as an environment variable for these. That is, depending on the **-e** setting, it may be overridden by a makefile definition.

*option* For a complete list and description of all make utility options, see Section 5.4, *Make Utility Options*.

### Exit status

The make utility returns an exit status of 1 when it halts as a result of an error. Otherwise it returns an exit status of 0.

## 4.2.2. Writing a Makefile

In addition to the standard makefile `makefile`, the make utility always reads the makefile `mkmcs.mk` before other inputs. This system makefile contains implicit rules and predefined macros that you can use in the makefile `makefile`.

With the option **-r** (Do not read the `mkmcs.mk` file) you can prevent the make utility from reading `mkmcs.mk`.

The default name of the makefile is `makefile` in the current directory. If you want to use another makefile, use the option **-f**.

The makefile can contain a mixture of:

- targets and dependencies
- rules
- macro definitions or functions
- conditional processing
- comment lines
- include lines
- export lines

To continue a line on the next line, terminate it with a backslash (****):

```
# this comment line is continued\
on the next line
```

If a line must end with a backslash, add an empty macro:

```
# this comment line ends with a backslash \$(EMPTY)
# this is a new line
```

### 4.2.2.1. Targets and Dependencies

The basis of the makefile is a set of targets, dependencies and rules. A target entry in the makefile has the following format:

```
target ... : [dependency ...] [; rule]
       [rule]
        ...
```

Target lines must always start at the beginning of a line, leading white spaces (tabs or spaces) are not allowed. A target line consists of one or more targets, a semicolon and a set of files which are required to build the target (*dependencies*). The target itself can be one or more filenames or symbolic names:

```
all:                    demo.elf final.elf

demo.elf final.elf:     test.o demo.o final.o
```

You can now can specify the target you want to build to the make utility. The following three invocations all have the same effect:

```
mkmcs
mkmcs all
mkmcs demo.elf final.elf
```

If you do *not* specify a target, the first target in the makefile (in this example `all`) is built. The target `all` depends on `demo.elf` and `final.elf` so the second and third invocation have the same effect and the files `demo.elf` and `final.elf` are built.

You can normally use colons to denote drive letters. The following works as intended:

```
c:foo.o : a:foo.asm
```

If a target is defined in more than one target line, the dependencies are added to form the target's complete dependency list:

```
all:  demo.elf   # These two lines are equivalent with:
all:  final.elf  # all: demo.elf final.elf
```

#### Special targets

There are a number of special targets. Their names begin with a period.

| Target | Description |
|---|---|
| `.DEFAULT` | If you call the make utility with a target that has no definition in the makefile, this target is built. |
| `.DONE` | When the make utility has finished building the specified targets, it continues with the rules following this target. |
| `.IGNORE` | Non-zero error codes returned from commands are ignored. Encountering this in a makefile is the same as specifying the option **-i** on the command line. |
| `.INIT` | The rules following this target are executed before any other targets are built. |
| `.PRECIOUS` | Dependency files mentioned for this target are never removed. Normally, if a command in a rule returns an error or when the target construction is interrupted, the make utility removes that target file. You can use the option **-p** on the command line to make all targets precious. |
| `.SILENT` | Commands are not echoed before executing them. Encountering this in a makefile is the same as specifying the option **-s** on the command line. |
| `.SUFFIXES` | This target specifies a list of file extensions. Instead of building a completely specified target, you now can build a target that has a certain file extension. Implicit rules to build files with a number of extensions are included in the system makefile `mkmcs.mk`.<br>If you specify this target with dependencies, these are added to the existing `.SUFFIXES` target in `mkmcs.mk`. If you specify this target without dependencies, the existing list is cleared. |

### 4.2.2.2. Makefile Rules

A line with leading white space (tabs or spaces) is considered as a rule and associated with the most recently preceding dependency line. A *rule* is a line with commands that are executed to build the associated target. A target-dependency line can be followed by one or more rules.

```
final.o   : final.asm                # target and dependency
            move test.asm final.asm  # rule1
            asmcs final.asm          # rule2
```

You can precede a rule with one or more of the following characters:

| | |
|---|---|
| @ | does not echo the command line, except if **-n** is used. |
| - | the make utility ignores the exit code of the command. Normally the make utility stops if a non-zero exit code is returned. This is the same as specifying the option **-i** on the command line or specifying the special `.IGNORE` target. |
| + | The make utility uses a shell or Windows command prompt (`cmd.exe`) to execute the command. If the '+' is not followed by a shell line, but the command is an MS-DOS command or if redirection is used (<, \|, >), the shell line is passed to `cmd.exe` anyway. |

You can force **mkmcs** to execute multiple command lines in one shell environment. This is accomplished with the token combination '**;**'. For example:

```
cd c:\Tasking\bin ;\
mkmcs -V
```

Note that the ';' must always directly be followed by the '\' token. Whitespace is not removed when it is at the end of the previous command line or when it is in front of the next command line. The use of the ';' as an operator for a command (like a semicolon ';' separated list with each item on one line) and the '\' as a layout tool is not supported, unless they are separated with whitespace.

## Inline temporary files

The make utility can generate inline temporary files. If a line contains **<<***LABEL* (no whitespaces!) then all subsequent lines are placed in a temporary file until the line *LABEL* is encountered. Next, **<<***LABEL* is replaced by the name of the temporary file. For example:

```
lmcs -o $@ -f <<EOF
     $(separate "\n" $(match .o $!))
     $(separate "\n" $(match .a $!))
     $(LKFLAGS)
EOF
```

The three lines between <<EOF and EOF are written to a temporary file (for example mkce4c0a.tmp), and the rule is rewritten as: lmcs -o $@ -f mkce4c0a.tmp.

## Suffix targets

Instead of specifying a specific target, you can also define a general target. A general target specifies the rules to generate a file with extension .ex1 to a file with extension .ex2. For example:

```
.SUFFIXES:  .asm
.asm.o     :
            ccmcs -c $<
```

Read this as: to build a file with extension .o out of a file with extension .asm, call the control program with **-c $<**. **$<** is a predefined macro that is replaced with the name of the current dependency file. The special target .SUFFIXES: is followed by a list of file extensions of the files that are required to build the target.

## Implicit rules

Implicit rules are stored in the system makefile mkmcs.mk and are intimately tied to the .SUFFIXES special target. Each dependency that follows the .SUFFIXES target, defines an extension to a filename which must be used to build another file. The implicit rules then define how to actually build one file from another. These files share a common basename, but have different extensions.

If the specified target on the command line is not defined in the makefile or has not rules in the makefile, the make utility looks if there is an implicit rule to build the target.

Example:

```
OPTS =     --map-file            # macro

prog.elf:  prog.o sub.o
     lmcs  prog.o sub.o $(OPTS) -o prog.elf

prog.o:    prog.asm myinc.inc
```

```
     asmcs prog.asm

sub.o:     sub.asm myinc.inc
     asmcs sub.asm
```

This makefile says that `prog.elf` depends on two files `prog.o` and `sub.o`, and that they in turn depend on their corresponding source files (`prog.asm` and `sub.asm`) along with the common file `myinc.inc`.

The following makefile uses implicit rules (from `mkmcs.mk`) to perform the same job.

```
LDFLAGS = --map-file           # macro used by implicit rules
prog.elf: prog.o sub.o         # implicit rule used
prog.o: prog.asm myinc.inc     # implicit rule used
sub.o:  sub.asm myinc.inc      # implicit rule used
```

### 4.2.2.3. Macro Definitions

A *macro* is a symbol name that is replaced with its definition before the makefile is executed. Although the macro name can consist of lowercase or uppercase characters, uppercase is an accepted convention. The general form of a macro definition is:

```
MACRO = text
MACRO += and more text
```

Spaces around the equal sign are not significant. With the **+=** operator you can add a string to an existing macro. An extra space is inserted before the added string automatically.

To use a macro, you must access its contents:

```
$(MACRO)       # you can read this as
${MACRO}       # the contents of macro MACRO
```

If the macro name is a single character, the parentheses are optional. Note that the expansion is done recursively, so the body of a macro may contain other macros. These macros are expanded when the macro is actually used, not at the point of definition:

```
FOOD = $(EAT) and $(DRINK)
EAT = meat and/or vegetables
DRINK = water
export FOOD
```

The macro `FOOD` is expanded as `meat and/or vegetables and water` at the moment it is used in the export line, and the environment variable FOOD is set accordingly.

#### Predefined macros

| Macro | Description |
|---|---|
| MAKE | Holds the value **mkmcs**. Any line which uses MAKE, temporarily overrides the option **-n** (Show commands without executing), just for the duration of the one line. This way you can test nested calls to MAKE with the option **-n**. |

| Macro | Description |
|---|---|
| `MAKEFLAGS` | Holds the set of options provided to **mkmcs** (except for the options **-f** and **-d**). If this macro is exported to set the environment variable MAKEFLAGS, the set of options is processed before any command line options. You can pass this macro explicitly to nested **mkmcs**'s, but it is also available to these invocations as an environment variable. |
| `PRODDIR` | Holds the name of the directory where **mkmcs** is installed. You can use this macro to refer to files belonging to the product, for example an include directory.<br>`INCDIR = $(PRODDIR)/include`<br>When **mkmcs** is installed in the directory `c:/Tasking/bin` this line expands to:<br>`INCDIR = c:/Tasking/include` |
| `SHELLCMD` | Holds the default list of commands which are local to the SHELL. If a rule is an invocation of one of these commands, a SHELL is automatically spawned to handle it. |
| $ | This macro translates to a dollar sign. Thus you can use "$$" in the makefile to represent a single "$". |

## Dynamically maintained macros

There are several dynamically maintained macros that are useful as abbreviations within rules. It is best not to define them explicitly.

| Macro | Description |
|---|---|
| `$*` | The basename of the current target. |
| $< | The name of the current dependency file. |
| $@ | The name of the current target. |
| $? | The names of dependents which are younger than the target. |
| `$!` | The names of all dependents. |

The $< and $* macros are normally used for implicit rules. They may be unreliable when used within explicit target command lines. All macros may be suffixed with **F** to specify the Filename components (e.g. `${*F}`, `${@F}`). Likewise, the macros $*, $< and $@ may be suffixed by **D** to specify the Directory component.

The result of the $* macro is always without double quotes ("), regardless of the original target having double quotes (") around it or not.

The result of using the suffix **F** (Filename component) or **D** (Directory component) is also always without double quotes ("), regardless of the original contents having double quotes (") around it or not.

### 4.2.2.4. Makefile Functions

A function not only expands but also performs a certain operation. Functions syntactically look like macros but have embedded spaces in the macro name, e.g. '$(match arg1 arg2 arg3 )'. All functions are built-in and currently these are: `match`, `separate`, `protect`, `exist`,`nexist` and `addprefix`.

#### $(match *suffix filename* ...)

The `match` function yields all arguments which match a certain suffix:

```
$(match .o prog.o sub.o mylib.a)
```

yields:

```
prog.o sub.o
```

#### $(separate *separator argument* ...)

The `separate` function concatenates its arguments using the first argument as the separator. If the first argument is enclosed in double quotes then '\n' is interpreted as a newline character, '\t' is interpreted as a tab, '*ooo*' is interpreted as an octal value (where, *ooo* is one to three octal digits), and spaces are taken literally. For example:

```
$(separate "\n" prog.o sub.o)
```

results in:

```
prog.o
sub.o
```

Function arguments may be macros or functions themselves. So,

```
$(separate "\n" $(match .o $!))
```

yields all object files the current target depends on, separated by a newline string.

#### $(protect *argument*)

The `protect` function adds one level of quoting. This function has one argument which can contain white space. If the argument contains any white space, single quotes, double quotes, or backslashes, it is enclosed in double quotes. In addition, any double quote or backslash is escaped with a backslash.

Example:

```
echo $(protect I'll show you the "protect" function)
```

yields:

```
echo "I'll show you the \"protect\" function"
```

#### $(exist *file* | *directory argument*)

The `exist` function expands to its second argument if the first argument is an existing file or directory.

Example:

```
$(exist test.asm ccmcs test.asm)
```

When the file `test.asm` exists, it yields:

```
ccmcs test.asm
```

When the file `test.asm` does not exist nothing is expanded.

#### $(nexist *file|directory argument*)

The `nexist` function is the opposite of the `exist` function. It expands to its second argument if the first argument is not an existing file or directory.

Example:

```
$(nexist test.o ccmcs test.asm)
```

#### $(addprefix *prefix*, *argument* ...)

The `addprefix` function adds a prefix to its arguments. It is used in `mkmcs.mk` for invocation of the control program to pass arguments directly to a tool.

Example:

```
ccmcs $(addprefix -Wa, -gs -k) test.asm
```

yields:

```
ccmcs -Wa-gs -Wa-k test.asm
```

### 4.2.2.5. Conditional Processing

Lines containing `ifdef`, `ifndef`, `else` or `endif` are used for conditional processing of the makefile. They are used in the following way:

```
ifdef macro-name
if-lines
else
else-lines
endif
```

The *if-lines* and *else-lines* may contain any number of lines or text of any kind, even other `ifdef`, `ifndef`, `else` and `endif` lines, or no lines at all. The `else` line may be omitted, along with the *else-lines* following it.

First the *macro-name* after the `ifdef` command is checked for definition. If the macro is defined then the *if-lines* are interpreted and the *else-lines* are discarded (if present). Otherwise the *if-lines* are discarded; and if there is an `else` line, the *else-lines* are interpreted; but if there is no else line, then no lines are interpreted.

When you use the `ifndef` line instead of `ifdef`, the macro is tested for not being defined. These conditional lines can be nested up to 6 levels deep.

You can also add tests based on strings. With `ifeq` the result is true if the two strings match, with `ifneq` the result is true if the two strings do not match. They are used in the following way:

```
ifeq(string1,string2)
if-lines
else
else-lines
endif
```

### 4.2.2.6. Comment, Include and Export Lines

#### Comment lines

Anything after a "#" is considered as a comment, and is ignored. If the "#" is inside a quoted string, it is not treated as a comment. Completely blank lines are ignored.

```
test.o  : test.asm        # this is comment and is
          ccmcs test.asm  # ignored by the make utility
```

#### Include lines

An *include line* is used to include the text of another makefile (like including a `.h` file in a C source). Macros in the name of the included file are expanded before the file is included. You can include several files. Include files may be nested.

```
include makefile2 makefile3
```

#### Export lines

An *export line* is used to export a macro definition to the environment of any command executed by the make utility.

```
GREETING = Hello
export GREETING
```

This example creates the environment variable `GREETING` with the value `Hello`. The macro is exported at the moment the export line is read so the macro definition has to precede the export line.

## 4.3. Make Utility amk

**amk** is the make utility Eclipse uses to maintain, update, and reconstruct groups of programs. But you can also use it on the command line. Its features are a little different from **mkmcs**. The main difference compared to **mkmcs** and other make utilities, is that **amk** features parallelism which utilizes the multiple cores found on modern host hardware, hardening for path names with embedded white space and it has an (internal) interface to provide progress information for updating a progress bar. It does not use an external command shell (`/bin/sh`, `cmd.exe`) but executes commands directly.

The primary purpose of any make utility is to speed up the edit-build-test cycle. To avoid having to build everything from scratch even when only one source file changes, it is necessary to describe dependencies between source files and output files and the commands needed for updating the output files. This is done in a so called "makefile".

### 4.3.1. Makefile Rules

A makefile dependency rule is a single line of the form:

```
[target ...] : [prerequisite ...]
```

where *target* and *prerequisite* are path names to files. Example:

```
test.o : test.asm
```

This states that target `test.o` depends on prerequisite `test.asm`. So, whenever the latter is modified the first must be updated. Dependencies accumulate: prerequisites and targets can be mentioned in multiple dependency rules (circular dependencies are not allowed however). The command(s) for updating a target when any of its prerequisites have been modified must be specified with leading white space after any of the dependency rule(s) for the target in question. Example:

```
test.o :
  ccmcs test.asm   # leading white space
```

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

```
test.o : test.asm
  ccmcs test.asm
```

White space around the colon is not required. When a path name contains special characters such as '**:**', '**#**' (start of comment), '**=**' (macro assignment) or any white space, then the path name must be enclosed in single or double quotes. Quoted strings can contain anything except the quote character itself and a newline. Two strings without white space in between are interpreted as one, so it is possible to embed single and double quotes themselves by switching the quote character.

When a target does not exist, its modification time is assumed to be very old. So, **amk** will try to make it. When a prerequisite does not exist possibly after having tried to make it, it is assumed to be very new. So, the update commands for the current target will be executed in that case. **amk** will only try to make targets which are specified on the command line. The default target is the first target in the makefile which does not start with a dot.

## Static pattern rules

Static pattern rules are rules which specify multiple targets and construct the prerequisite names for each target based on the target name.

```
[target ...] : target-pattern : [prerequisite-patterns ...]
```

The *target* specifies the targets the rules applies to. The *target-pattern* and *prerequisite-patterns* specify how to compute the prerequisites of each target. Each target is matched against the *target-pattern* to extract a part of the target name, called the *stem*. This stem is substituted into each of the *prerequisite-patterns* to make the prerequisite names (one from each *prerequisite-pattern*).

Each pattern normally contains the character '%' just once. When the *target-pattern* matches a target, the '%' can match any part of the target name; this part is called the *stem*. The rest of the pattern must match exactly. For example, the target `foo.o` matches the pattern '`%.o`', with '`foo`' as the stem. The targets `foo.asm` and `foo.elf` do not match that pattern.

The prerequisite names for each target are made by substituting the stem for the '%' in each prerequisite pattern.

Example:

```
objects = test.o filter.o

all: $(objects)

$(objects): %.o: %.asm
    ccmcs -c $< -o $@
    echo the stem is $*
```

Here '$<' is the automatic variable that holds the name of the prerequisite, '$@' is the automatic variable that holds the name of the target and '`$*`' is the stem that matches the pattern. Internally this translates to the following two rules:

```
test.o: test.asm
    ccmcs -c test.asm -o test.o
    echo the stem is test

filter.o: filter.asm
    ccmcs -c filter.asm -o filter.o
    echo the stem is filter
```

Each target specified must match the target pattern; a warning is issued for each target that does not.

## Special targets

There are a number of special targets. Their names begin with a period.

| Target | Description |
|---|---|
| `.DEFAULT` | If you call the make utility with a target that has no definition in the makefile, this target is built. |

| Target | Description |
|---|---|
| .DONE | When the make utility has finished building the specified targets, it continues with the rules following this target. |
| .INIT | The rules following this target are executed before any other targets are built. |
| .PHONY | The prerequisites of this target are considered to be phony targets. A phony target is a target that is not really the name of a file. The rules following a phony target are executed unconditionally, regardless of whether a file with that name exists or what its last-modification time is.<br>For example:<br>`.PHONY: clean`<br>`clean:`<br>`        rm *.o`<br>With `amk clean`, the command is executed regardless of whether there is a file named `clean`. |

## 4.3.2. Makefile Directives

Directives inside makefiles are executed while reading the makefile. When a line starts with the word "`include`" or "`-include`" then the remaining arguments on that line are considered filenames whose contents are to be inserted at the current line. "`-include`" will silently skip files which are not present. You can include several files. Include files may be nested.

Example:

```
include makefile2 makefile3
```

White spaces (tabs or spaces) in front of the directive are allowed.

## 4.3.3. Macro Definitions

A *macro* is a symbol name that is replaced with its definition before the makefile is executed. Although the macro name can consist of lowercase or uppercase characters, uppercase is an accepted convention. When a line does not start with white space and contains the assignment operator '**=**', '**:=**' or '**+=**' then the line is interpreted as a macro definition. White space around the assignment operator and white space at the end of the line is discarded. Single character macro evaluation happens by prefixing the name with '**$**'. To evaluate macros with names longer than one character put the name between parentheses '**()**' or curly braces '**{}**'. Macro names may contain anything, even white space or other macro evaluations. Example:

```
DINNER = $(FOOD) and $(BEVERAGE)
FOOD = pizza
BEVERAGE = sparkling water
FOOD += with cheese
```

With the **+=** operator you can add a string to an existing macro. An extra space is inserted before the added string automatically.

Macros are evaluated recursively. Whenever `$(DINNER)` or `${DINNER}` is mentioned after the above, it will be replaced by the text "`pizza with cheese and sparkling water`". The left hand side in a macro definition is evaluated before the definition takes place. Right hand side evaluation depends on the assignment operator:

| | |
|---|---|
| = | Evaluate the macro at the moment it is used. |
| `:=` | Evaluate the replacement text before defining the macro. |

Subsequent '**+=**' assignments will inherit the evaluation behavior from the previous assignment. If there is none, then '**+=**' is the same as '**=**'. The default value for any macro is taken from the environment. Macro definitions inside the makefile overrule environment variables. Macro definitions on the **amk** command line will be evaluated first and overrule definitions inside the makefile.

| Macro | Description |
|---|---|
| `$` | This macro translates to a dollar sign. Thus you can use "$$" in the makefile to represent a single "$". |
| `@` | The name of the current target. When a rule has multiple targets, then it is the name of the target that caused the rule commands to be run. |
| `*` | The basename (or stem) of the current target. The stem is either provided via a static pattern rule or is calculated by removing all characters found after and including the last dot in the current target name. If the target name is '`test.asm`' then the stem is '`test`' (if the target was not created via a static pattern rule). |
| `<` | The name of the first prerequisite. |
| `MAKE` | The **amk** path name (quoted if necessary). Optionally followed by the options **-n** and **-s**. |
| `ORIGIN` | The name of the directory where **amk** is installed (quoted if necessary). |
| `SUBDIR` | The argument of option **-G**. If you have nested makes with **-G** options, the paths are combined. This macro is defined in the environment (i.e. default macro value). |

The @, * and < macros may be suffixed by '**D**' to specify the directory component or by '**F**' to specify the filename component. `$(@D)` evaluates to the directory name holding the file`$(@F)`. `$(@D)/$(@F)` is equivalent to `$@`. Note that on MS-Windows most programs accept forward slashes, even for UNC path names.

The result of the predefined macros @, * and < and 'D' and 'F' variants is not quoted, so it may be necessary to put quotes around it.

Note that stem calculation can cause unexpected values. For example:

```
$@                     $*
/home/.wine/test       /home/
/home/test/.project    /home/test/
/../file               /.
```

## Macro string substitution

When the macro name in an evaluation is followed by a colon and equal sign as in

```
$(MACRO:string1=string2)
```

then **amk** will replace *string1* at the end of every word in `$(MACRO)` by *string2* during evaluation. When `$(MACRO)` contains quoted path names, the quote character must be mentioned in both the original string and the replacement string[1]. For example:

```
$(MACRO:.o"=.d")
```

[1] Internally, **amk** tokenizes the evaluated text, but performs substitution on the original input text to preserve compatibility here with existing make implementations and POSIX.

### 4.3.4. Makefile Functions

A function not only expands but also performs a certain operation. The following functions are available:

#### $(filter *pattern* ...,*item* ...)

The `filter` function filters a list of items using a pattern. It returns *items* that do match any of the *pattern* words, removing any items that do not match. The patterns are written using '%',

```
    ${filter %.asm %.inc, test.asm test.inc test.o readme.txt .project output.asm}
```

results in:

```
    test.asm test.inc output.asm
```

#### $(filter-out *pattern* ...,*item* ...)

The `filter-out` function returns all *items* that do not match any of the *pattern* words, removing the items that do match one or more. This is the exact opposite of the `filter` function.

```
    ${filter-out %.asm %.inc, test.asm test.inc test.o readme.txt .project output.asm}
```

results in:

```
    test.o readme.txt .project
```

#### $(foreach *var-name*, *item* ..., *action*)

The `foreach` function runs through a list of items and performs the same *action* for each *item*. The *var-name* is the name of the macro which gets dynamically filled with an item while iterating through the *item* list. In the *action* you can refer to this macro. For example:

```
    ${foreach T, test filter output, ${T}.asm ${T}.inc}
```

results in:

```
    test.asm test.inc filter.asm filter.inc output.asm output.inc
```

### 4.3.5. Conditional Processing

Lines containing `ifdef`, `ifndef`, `else` or `endif` are used for conditional processing of the makefile. They are used in the following way:

```
ifdef macro-name
if-lines
else
else-lines
endif
```

The *if-lines* and *else-lines* may contain any number of lines or text of any kind, even other `ifdef`, `ifndef`, `else` and `endif` lines, or no lines at all. The `else` line may be omitted, along with the *else-lines* following it. White spaces (tabs or spaces) in front of preprocessing directives are allowed.

First the *macro-name* after the `ifdef` command is checked for definition. If the macro is defined then the *if-lines* are interpreted and the *else-lines* are discarded (if present). Otherwise the *if-lines* are discarded; and if there is an `else` line, the *else-lines* are interpreted; but if there is no else line, then no lines are interpreted.

When you use the `ifndef` line instead of `ifdef`, the macro is tested for not being defined. These conditional lines can be nested to any level.

You can also add tests based on strings. With `ifeq` the result is true if the two strings match, with `ifneq` the result is true if the two strings do not match. They are used in the following way:

```
ifeq(string1,string2)
if-lines
else
else-lines
endif
```

## 4.3.6. Makefile Parsing

**amk** reads and interprets a makefile in the following order:

1. When the last character on a line is a backslash (****) (i.e. without trailing white space) then that line and the next line will be concatenated, removing the backslash and newline.

2. The unquoted '**#**' character indicates start of comment and may be placed anywhere on a line. It will be removed in this phase.

   ```
   # this comment line is continued\
   on the next line
   ```

3. Trailing white space is removed.

4. When a line starts with white space and it is not followed by a directive or preprocessing directive, then it is interpreted as a command for updating a target.

5. Otherwise, when a line contains the unquoted text '**=**', '**+=**' or '**:=**' operator, then it will be interpreted as a macro definition.

6. Otherwise, all macros on the line are evaluated before considering the next steps.

7. When the resulting line contains an unquoted '**:**' the line is interpreted as a dependency rule.

8. When the first token on the line is "`include`" or "`-include`" (which by now must start on the first column of the line), **amk** will execute the directive.

9. Otherwise, the line must be empty.

Macros in commands for updating a target are evaluated right before the actual execution takes place (or would take place when you use the **-n** option).

## 4.3.7. Makefile Command Processing

A line with leading white space (tabs or spaces) without a (preprocessing) directive is considered as a command for updating a target. When you use the option **-j** or **-J**, **amk** will execute the commands for updating different targets in parallel. In that case standard input will not be available and standard output and error output will be merged and displayed on standard output only after the commands have finished for a target.

You can precede a command by one or more of the following characters:

| | |
|---|---|
| @ | Do not show the command. By default, commands are shown prior to their output. |
| - | Continue upon error. This means that **amk** ignores a non-zero exit code of the command. |
| + | Execute the command, even when you use option **-n** (dry run). |
| \| | Execute the command on the foreground with standard input, standard output and error output available. |

### Built-in commands

| Command | Description |
|---|---|
| `true` | This command does nothing. Arguments are ignored. |
| `false` | This command does nothing, except failing with exit code 1. Arguments are ignored. |
| `echo` *arg*... | Display a line of text. |
| `exit` *code* | Exit with defined code. Depending on the program arguments and/or the extra rule options '-' this will cause **amk** to exit with the provided code. Please note that '`exit 0`' has currently no result. |
| `argfile` *file arg*... | Create an argument file suitable for the **--option-file** (**-f**) option of all the other tools. The first `argfile` argument is the name of the file to be created. Subsequent arguments specify the contents. An existing argument file is not modified unless necessary. So, the argument file itself can be used to create a dependency to options of the command for updating a target. |
| `rm` [*option*]... *file*... | Remove the specified file(s). The following options are available:<br>**-r**, **--recursive** Remove directories and their contents recursively.<br>**-f**, **--force** Force deletion. Ignore non-existent files, never prompt.<br>**-i**, **--interactive** Interactive. Prompt before every removal.<br>**-v**, **--verbose** Verbose mode. Explain what is being done.<br>**-m** *file* Read options from *file*..<br>**-?**, **--help** Show usage. |

## 4.3.8. Calling the amk Make Utility

The invocation syntax of **amk** is:

```
amk [option]... [target]... [macro=def]...
```

For example:

```
amk test.elf
```

| | |
|---|---|
| *target* | You can specify any target that is defined in the makefile. A target can also be one of the intermediate files specified in the makefile. |
| *macro*=*def* | Macro definition. This definition remains fixed for the **amk** invocation. It overrides any regular definitions for the specified macro within the makefiles and from the environment. It is not inherited by subordinate **amk**'s |
| *option* | For a complete list and description of all **amk** make utility options, see Section 5.5, *Parallel Make Utility Options*. |

### Exit status

The make utility returns an exit status of 1 when it halts as a result of an error. Otherwise it returns an exit status of 0.

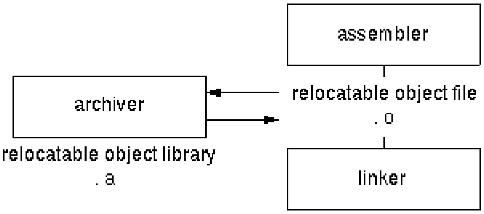
## 4.4. Archiver

The archiver **armcs** is a program to build and maintain your own library files. A library file is a file with extension `.a` and contains one or more object files (`.o`) that may be used by the linker.

The archiver has five main functions:

- Deleting an object module from the library
- Moving an object module to another position in the library file
- Replacing an object module in the library or add a new object module
- Showing a table of contents of the library file
- Extracting an object module from the library

The archiver takes the following files for input and output:



The linker optionally includes object modules from a library if that module resolves an external symbol definition in one of the modules that are read before.

## 4.4.1. Calling the Archiver

You can create a library in Eclipse, which calls the archiver or you can call the archiver on the command line.

### To create a library in Eclipse

Instead of creating an MCS absolute ELF file, you can choose to create a library. You do this when you create a new project with the New Assembly Project wizard. (**File »** ) select the option  in the following dialog.

1. From the **File** menu, select **New » TASKING MCS Assembly Project**.

   *The New Assembly Project wizard appears.*

2. Enter a project name.

3. In the **Project type** box, select **TASKING MCS Library** and click**Next >**.

4. Follow the rest of the wizard and click **Finish**.

5. Add the files to your project.

6. Build the project as usual. For example, select **Project » Build Project** (101).

   *Eclipse builds the library. Instead of calling the linker, Eclipse now calls the archiver.*

### Command line invocation

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

```
armcs key_option [sub_option...] library [object_file]
```

*key_option* With a key option you specify the main task which the archiver should perform. You must *always* specify a key option.

*sub_option* Sub-options specify into more detail how the archiver should perform the task that is specified with the key option. It is not obligatory to specify sub-options.

*library* The name of the library file on which the archiver performs the specified action. You must always specify a library name, except for the options **-?** and **-V**. When the library is not in the current directory, specify the complete path (either absolute or relative) to the library.

*object_file* The name of an object file. You must always specify an object file name when you add, extract, replace or remove an object file from the library.

### Options of the archiver utility

The following archiver options are available:

| Description | Option | Sub-option |
|---|---|---|
| **Main functions (key options)** | | |
| Replace or add an object module | **-r** | **-a -b -c -u -v** |
| Extract an object module from the library | **-x** | **-v** |
| Delete object module from library | **-d** | **-v** |
| Move object module to another position | **-m** | **-a -b -v** |
| Print a table of contents of the library | **-t** | **-s0 -s1** |
| Print object module to standard output | **-p** | |
| **Sub-options** | | |
| Append or move new modules after existing module *name* | **-a** *name* | |
| Append or move new modules before existing module *name* | **-b** *name* | |
| Create library without notification if library does not exis | **-c** | |
| Preserve last-modified date from the library | **-o** | |
| Print symbols in library modules | **-s**{**0**\|**1**} | |
| Replace only newer modules | **-u** | |
| Verbose | **-v** | |
| **Miscellaneous** | | |
| Display options | **-?** | |
| Display version header | **-V** | |
| Read options from *file* | **-f** *file* | |
| Suppress warnings above level *n* | **-w***n* | |

For a complete list and description of all archiver options, see Section 5.6, *Archiver Options*.

## 4.4.2. Archiver Examples

### Create a new library

If you add modules to a library that does not yet exist, the library is created. To create a new library with the name `mylib.a` and add the object modules `cstart.o` and `calc.o` to it:

```
armcs -r mylib.a cstart.o calc.o
```

### Add a new module to an existing library

If you add a new module to an existing library, the module is added at the end of the module. (If the module already exists in the library, it is replaced.)

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

### Print a list of object modules in the library

To inspect the contents of the library:

```
armcs -t mylib.a
```

The library has the following contents:

```
cstart.o
calc.o
mod3.o
```

### Move an object module to another position

To move `mod3.o` to the beginning of the library, position it just before `cstart.o`:

```
armcs -mb cstart.o mylib.a mod3.o
```

### Delete an object module from the library

To delete the object module `cstart.o` from the library `mylib.a`:

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

### Extract all modules from the library

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

```
armcs -x mylib.a
```

## 4.5. Bosch MCS Assembly to TASKING Assembly Converter

The Perl script **cnvba2ta.pl** is a converter to convert Bosch MCS assembly files to TASKING VX-toolset for MCS assembly files. You need to have Perl installed on your system.

### Command line invocation

You can call the converter from the command line by using Perl. The invocation syntax is:

**`perl cnvba2ta.pl`** *`input_file`* **`>`** *`output_file`*

Without output redirection the output is sent to `stdout`. The converted file includes `mcs_defines.inc`.

For example:

```
perl cnvba2ta.pl bosch_mcs.mcs > tsk_mcs.asm
```

# Chapter 5. Tool Options

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

## Tool options in Eclipse (Menu entry)

For each tool option that you can set from within Eclipse, a **Menu entry** description is available. In Eclipse you can customize the tools and tool options in the following dialog:

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

   *The Properties dialog appears*.

2. In the left pane, expand **C/C++ Build** and select **Settings**.

   *In the right pane the Settings appear.*

3. Open the **Tool Settings** tab.

   *You can set all tool options here.*

> Unless stated otherwise, all **Menu entry** descriptions expect that you have this Tool Settings tab open.

The following tables give an overview of all tool options on the Tool Settings tab in Eclipse with hyperlinks to the corresponding command line options (if available).

## Global Options

| Eclipse option | Description or option |
|---|---|
| Use global 'product directory' preference | Directory where the TASKING toolset is installed |
| Treat warnings as errors | Control program option **--warnings-as-errors** |
| Keep temporary files | Control program option **--keep-temporary-files (-t)** |
| Verbose mode of control program | Control program option **--verbose (-v)** |

## Assembler

| Eclipse option | Description or option |
|---|---|
| **Preprocessing** | |
| Defined symbols | Assembler option **--define** |
| Pre-include files | Assembler option **--include-file** |

| Eclipse option | Description or option |
|---|---|
| **Include Paths** | |
| Include paths | Assembler option **--include-directory** |
| **Symbols** | |
| Generate symbolic debug | Assembler option **--debug-info** |
| Case insensitive identifiers | Assembler option **--case-insensitive** |
| Emit local EQU symbols | Assembler option **--emit-locals=+equ** |
| Emit local non-EQU symbols | Assembler option **--emit-locals=+symbols** |
| Set default symbol scope to global | Assembler option **--symbol-scope** |
| **Optimization** | |
| Optimize instruction size | Assembler option **--optimize=+instr-size** |
| **List File** | |
| Generate list file | Control program option **--list-files** |
| List ... | Assembler option **--list-format** |
| List section summary | Assembler option **--section-info=+list** |
| **Diagnostics** | |
| Suppress warnings | Assembler option **--no-warnings=***num* |
| Suppress all warnings | Assembler option **--no-warnings** |
| Display section summary | Assembler option **--section-info=+console** |
| Maximum number of emitted errors | Assembler option **--error-limit** |
| **Miscellaneous** | |
| Additional options | Assembler options |

## Linker

| Eclipse option | Description or option |
|---|---|
| **Libraries** | |
| Rescan libraries to solve unresolved externals | Linker option **--no-rescan** |
| Libraries | The libraries are added as files on the command line. |
| Library search path | Linker option **--library-directory** |
| **Data Objects** | |
| Data objects | Linker option **--import-object** |
| **Script File** | |
| Defined symbols | Linker option **--define** |
| Linker script file (.lsl) | Linker option **--lsl-file** |
| **Optimization** | |

| Eclipse option | Description or option |
|---|---|
| Delete unreferenced sections | Linker option **--optimize=c** |
| Use a 'first-fit decreasing' algorithm | Linker option **--optimize=l** |
| Compress copy table | Linker option **--optimize=t** |
| Delete duplicate code | Linker option **--optimize=x** |
| Delete duplicate data | Linker option **--optimize=y** |
| **Map File** | |
| Generate map file (.map) | Control program option **--no-map-file** |
| Generate XML map file format (.mapxml) for map file viewer | Linker option **--map-file=***file***.mapxml:XML** |
| Include ... | Linker option **--map-file-format** |
| **Diagnostics** | |
| Suppress warnings | Linker option **--no-warnings=***num* |
| Suppress all warnings | Linker option **--no-warnings** |
| Maximum number of emitted errors | Linker option **--error-limit** |
| **Miscellaneous** | |
| Strip symbolic debug information | Linker option **--strip-debug** |
| Link case insensitive | Linker option **--case-insensitive** |
| Do not use standard copy table for initialization | Linker option **--user-provided-initialization-code** |
| Additional options | Linker options |

## 5.1. Assembler Options

This section lists all assembler options.

### Options in Eclipse versus options on the command line

Most command line options have an equivalent option in Eclipse but some options are only available on the command line. Eclipse invokes the assembler via the control program. Therefore, it uses the syntax of the control program to pass options and files to the assembler. If there is no equivalent option in Eclipse, you can specify a command line option in Eclipse as follows:

1. From the **Project** menu, select **Properties for**

   *The Properties dialog appears*.

2. In the left pane, expand **C/C++ Build** and select **Settings**.

   *In the right pane the Settings appear.*

3. On the Tool Settings tab, select **Assembler » Miscellaneous**.

4. In the **Additional options** field, enter one or more command line options.

   *Because Eclipse uses the control program, Eclipse automatically precedes the option with **-Wa** to pass the option via the control program directly to the assembler.*

Be aware that some command line options are not useful in Eclipse or just do not have any effect. For example, the option **-V** displays version header information and has no effect in Eclipse.

## Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (**-**) character, long option names always begin with two minus (**--**) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

Options can have flags or suboptions. To switch a flag 'on', use a lowercase letter or a **+***longflag*. To switch a flag off, use an uppercase letter or a **-***longflag*. Separate *longflags* with commas. The following two invocations are equivalent:

```
asmcs -Ogs test.asm
asmcs --optimize=+generics,+instr-size test.asm
```

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

# Assembler option: --case-insensitive (-c)

## Menu entry

1. Select **Assembler » Symbols**.

2. Enable the option **Case insensitive identifiers**.

## Command line syntax

**`--case-insensitive`**

**`-c`**

Default: case sensitive

## Description

With this option you tell the assembler not to distinguish between uppercase and lowercase characters. By default the assembler considers uppercase and lowercase characters as different characters.

## Example

When assembling case insensitive, the label `LabelName` is the same label as `labelname`.

```
asmcs --case-insensitive test.asm
```

## Related information

Assembler control **$CASE**

# Assembler option: --check

## Menu entry

-

## Command line syntax

```
--check
```

## Description

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

The assembler reports any warnings and/or errors.

This option is available on the command line only.

## Related information

-

# Assembler option: --cpu (-C)

## Menu entry

1. Expand **C/C++ Build** and select **Processor**.
2. From the **Processor Selection** list, select a processor or select **User defined TriCore ...**.

## Command line syntax

**`--cpu=`***`cpu`*

**`-C`***`cpu`*

## Description

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

## Example

To assemble the file `test.asm` for the TC27X processor:

```
asmcs --cpu=tc27x test.asm
```

## Related information

-

# Assembler option: --debug-info (-g)

## Menu entry

1. Select **Assembler » Symbols**.
2. Select an option from the **Generate symbolic debug** list.

## Command line syntax

**`--debug-info`**`[=`*`flags`*`]`

**`-g`**`[`*`flags`*`]`

You can set the following flags:

| | | |
|---|---|---|
| **+/-asm** | **a/A** | Assembly source line information |
| **+/-local** | **l/L** | Assembler local symbols debug information |

Default: **`--debug-info=-asm,-local`**

Default (without flags): **`--debug-info=+asm,+local`**

## Description

With this option you tell the assembler which kind of debug information to emit in the object file.

With **--debug-info=+asm** the assembler generates assembly source line information.

With **--debug-info=+local** the assembler generates local symbols debug information.

By default the assembler does not generate any debug information.

## Related information

Assembler control **$DEBUG**

# Assembler option: --define (-D)

## Menu entry

1. Select **Assembler » Preprocessing**.

   *The Defined symbols box right-below shows the symbols that are currently defined.*

2. To define a new symbol, click on the **Add** button in the **Defined symbols** box.

3. Type the symbol definition (for example, `demo=1`)

Use the **Edit** and **Delete** button to change a macro definition or to remove a macro from the list.

## Command line syntax

**`--define=`***`macro_name`*`[`**`=`***`macro_definition`*`]`

**`-D`***`macro_name`*`[`**`=`***`macro_definition`*`]`

## Description

With this option you can define a macro and specify it to the assembler preprocessor. If you only specify a macro name (no macro definition), the macro expands as '1'.

You can specify as many macros as you like. Simply use the **Add** button to add new macro definitions.

On the command line, use the option **--define** (**-D**) multiple times. If the command line exceeds the limit of the operating system, you can define the macros in an option file which you then must specify to the assembler with the option **--option-file** (**-f**) *file*.

Defining macros with this option (instead of in the assembly source) is, for example, useful in combination with conditional assembly as shown in the example below.

This option has the same effect as defining symbols via the `.DEFINE`, `.SET`, and `.EQU` directives. (similar to `#define` in the C language). With the `.MACRO` directive you can define more complex macros.

## Example

Consider the following assembly program with conditional code to assemble a demo program and a real program:

```
.IF DEMO == 1
...         ; instructions for demo application
.ELSE
...         ; instructions for the real application
.ENDIF
```

You can now use a macro definition to set the DEMO flag:

```
asmcs --define=DEMO test.asm
asmcs --define=DEMO=1 test.asm
```

Note that both invocations have the same effect.

## Related information

Assembler option **--option-file** (Specify an option file)

## Assembler option: --dep-file

### Menu entry

-

### Command line syntax

**`--dep-file`**`[=`*`file`*`]`

### Description

With this option you tell the assembler to generate dependency lines that can be used in a Makefile. The dependency information will be generated in addition to the normal output file.

By default, the information is written to a file with extension `.d`. When you specify a filename, all dependencies will be combined in the specified file.

### Example

```
asmcs --dep-file=test.dep test.asm
```

The assembler assembles the file `test.asm`, which results in the output file `test.o`, and generates dependency lines in the file `test.dep`.

### Related information

Assembler option **--make-target** (Specify target name for **--dep-file** output)

# Assembler option: --diag

## Menu entry

1. From the **Window** menu, select **Show View » Other » TASKING » Problems**.

   *The Problems view is added to the current perspective.*

2. In the Problems view right-click on a message.

   *A popup menu appears.*

3. Select **Detailed Diagnostics Info**.

   *A dialog box appears with additional information.*

## Command line syntax

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

You can set the following output formats:

| | |
|---|---|
| **html** | HTML output. |
| **rtf** | Rich Text Format. |
| **text** | ASCII text. |

Default format: text

## Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to stdout (normally your screen) and in the format you specify. You can specify the following formats: **html**, **rtf** or **text** (default). To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas.

## Example

To display an explanation of message number 244, enter:

```
asmcs --diag=244
```

This results in the following message and explanation:

```
W244: additional input files will be ignored

The assembler supports only a single input file. All other input files are ignored.
```

To write an explanation of all errors and warnings in HTML format to file `aserrors.html`, use redirection and enter:

```
asmcs --diag=html:all > aserrors.html
```

## Related information

Section 2.6, *Assembler Error Messages*

# Assembler option: --dwarf-version

## Menu entry

-

## Command line syntax

**`--dwarf-version=`**{**`2`**|**`3`**}

Default: **`3`**

## Description

With this option you tell the assembler which DWARF debug version to generate, DWARF2 or DWARF3 (default).

## Related information

-

# Assembler option: --emit-locals

## Menu entry

1. Select **Assembler » Symbols**.
2. Enable or disable one or both of the following options:
   - Emit local EQU symbols
   - Emit local non-EQU symbols

## Command line syntax

```
--emit-locals[=flag,...]
```

You can set the following flags:

| | | |
|---|---|---|
| **+/-equs** | **e/E** | emit local EQU symbols |
| **+/-symbols** | **s/S** | emit local non-EQU symbols |

Default: **`--emit-locals=ES`**

Default (without flags): **`--emit-locals=+symbols`**

## Description

With the option **--emit-locals=+equs** the assembler also emits local EQU symbols to the object file. Normally, only global symbols and non-EQU local symbols are emitted. Having local symbols in the object file can be useful for debugging.

## Related information

Assembler directive **.EQU**

# Assembler option: --error-file

## Menu entry

-

## Command line syntax

**`--error-file`**`[=`*`file`*`]`

## Description

With this option the assembler redirects error messages to a file. If you do not specify a filename, the error file will be named after the input file with extension `.ers`.

## Example

To write errors to `errors.ers` instead of `stderr`, enter:

```
asmcs --error-file=errors.ers test.asm
```

## Related information

Section 2.6, *Assembler Error Messages*

## Assembler option: --error-limit

### Menu entry

1. Select **Assembler » Diagnostics**.
2. Enter a value in the **Maximum number of emitted errors** field.

### Command line syntax

**`--error-limit=`***`number`*

Default: 42

### Description

With this option you tell the assembler to only emit the specified maximum number of errors. When 0 (null) is specified, the assembler emits all errors. Without this option the maximum number of errors is 42.

### Related information

Section 2.6, *Assembler Error Messages*

# Assembler option: --help (-?)

## Menu entry

-

## Command line syntax

**`--help`**`[`**`=`**`*item*]`

**`-?`**

You can specify the following arguments:

| | |
|---|---|
| **options** | Show extended option descriptions |

## Description

Displays an overview of all command line options. When you specify the argument **options** you can list detailed option descriptions.

## Example

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

```
asmcs -?
asmcs --help
asmcs
```

To see a detailed description of the available options, enter:

```
asmcs --help=options
```

## Related information

-

## Assembler option: --include-directory (-I)

### Menu entry

1. Select **Assembler » Include Paths**.

   *The Include paths box shows the directories that are added to the search path for include files.*

2. To define a new directory for the search path, click on the **Add** button in the **Include paths** box.

3. Type or select a path.

Use the **Edit** and **Delete** button to change a path or to remove a path from the list.

### Command line syntax

**`--include-directory=`***`path`*`,...`

**`-I`***`path`*`,...`

### Description

With this option you can specify the path where your include files are located. A relative path will be relative to the current directory,

The order in which the assembler searches for include files is:

1. The pathname in the assembly file and the directory of the assembly source.

2. The path that is specified with this option.

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

4. The default directory `$(PRODDIR)\include`.

### Example

Suppose that the assembly source file `test.asm` contains the following lines:

```
.INCLUDE 'myinc.inc'
```

You can call the assembler as follows:

```
asmcs --include-directory=c:\proj\include test.asm
```

First the assembler looks for the file `myinc.inc` in the directory where `test.asm` is located. If it does not find the file, it looks in the directory `c:\proj\include` (this option). If the file is still not found, the assembler searches in the environment variable and then in the default include directory.

## Related information

Assembler option **--include-file** (Include file at the start of the input file)

# Assembler option: --include-file (-H)

## Menu entry

1. Select **Assembler » Preprocessing**.

   *The Pre-include files box shows the files that are currently included before the assembling starts.*

2. To define a new file, click on the **Add** button in the **Pre-include files** box.

3. Type the full path and file name or select a file.

> Use the **Edit** and **Delete** button to change a file name or to remove a file from the list.

## Command line syntax

**`--include-file=`***`file`*`,...`

**`-H`***`file`*`,...`

## Description

With this option (set at project level) you include one extra file at the beginning of the assembly source file. The specified include file is included before all other includes. This is the same as specifying `.INCLUDE 'file'` at the beginning of your assembly source.

## Example

```
asmcs --include-file=myinc.inc test.asm
```

The file `myinc.inc` is included at the beginning of `test.asm` before it is assembled.

## Related information

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

## Assembler option: --keep-output-files (-k)

### Menu entry

Eclipse *always* removes the object file when errors occur during assembling.

### Command line syntax

**`--keep-output-files`**

**`-k`**

### Description

If an error occurs during assembling, the resulting object file (`.o`) may be incomplete or incorrect. With this option you keep the generated object file when an error occurs.

By default the assembler removes the generated object file when an error occurs. This is useful when you use the make utility. If the erroneous files are not removed, the make utility may process corrupt files on a subsequent invocation.

Use this option when you still want to use the generated object. For example when you know that a particular error does not result in a corrupt object file.

### Related information

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

# Assembler option: --list-file (-l)

## Menu entry

1. Select **Assembler » List File**.
2. Enable the option **Generate list file**.
3. Enable or disable the types of information to be included.

## Command line syntax

**`--list-file`**`[=`*`file`*`]`

**`-l`**`[`*`file`*`]`

Default: no list file is generated

## Description

With this option you tell the assembler to generate a list file. A list file shows the generated object code and the relative addresses. Note that the assembler generates a relocatable object file with relative addresses.

With the optional *file* you can specify an alternative name for the list file. By default, the name of the list file is the basename of the source file with the extension `.lst`.

## Related information

Assembler option **--list-format** (Format list file)

# Assembler option: --list-format (-L)

## Menu entry

1. Select **Assembler » List File**.
2. Enable the option **Generate list file**.
3. Enable or disable the types of information to be included.

## Command line syntax

**`--list-format=`***`flag`*`,...`

**`-L`***`flags`*

You can set the following flags:

| | | |
|---|---|---|
| **+/-section** | **d/D** | List section directives (`.SDECL`, `.SECT`) |
| **+/-symbol** | **e/E** | List symbol definition directives |
| **+/-macro** | **m/M** | List macro definitions |
| **+/-empty-line** | **n/N** | List empty source lines and comment lines |
| **+/-conditional** | **p/P** | List conditional assembly |
| **+/-equate** | **q/Q** | List equate and set directives (`.EQU`, `.SET`) |
| **+/-relocations** | **r/R** | List relocations characters 'r' |
| **+/-equate-values** | **v/V** | List equate and set values |
| **+/-wrap-lines** | **w/W** | Wrap source lines |
| **+/-macro-expansion** | **x/X** | List macro expansions |
| **+/-cycle-count** | **y/Y** | List cycle counts |
| **+/-define-expansion** | **z/Z** | List define expansions |

Use the following options for predefined sets of flags:

| | | |
|---|---|---|
| **--list-format=0** | **-L0** | All options disabled<br>Alias for **--list-format=DEMNPQRVWXYZ** |
| **--list-format=1** | **-L1** | All options enabled<br>Alias for **--list-format=demnpqrvwxyz** |

Default: **`--list-format=dEMnPqrVwXyZ`**

## Description

With this option you specify which information you want to include in the list file.

On the command line you must use this option in combination with the option **--list-file** (**-l**).

## Related information

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

Assembler option **--section-info=+list** (Display section information in list file)

## Assembler option: --make-target

### Menu entry

-

### Command line syntax

**`--make-target=`***`name`*

### Description

With this option you can overrule the default target name in the make dependencies generated by the option **--dep-file**. The default target name is the basename of the input file, with extension `.o`.

### Example

```
asmcs --dep-file --make-target=../mytarget.o test.asm
```

The assembler generates dependency lines with the default target name `../mytarget.o` instead of `test.o`.

### Related information

Assembler option **--dep-file** (Generate dependencies in a file)

# Assembler option: --no-warnings (-w)

## Menu entry

1. Select **Assembler » Diagnostics**.

   *The Suppress warnings box shows the warnings that are currently suppressed.*

2. To suppress a warning, click on the **Add** button in the **Suppress warnings** box.

3. Enter the numbers, separated by commas, of the warnings you want to suppress (for example `201,202`). Or you can use the **Add** button multiple times.

4. To suppress all warnings, enable the option **Suppress all warnings**.

Use the **Edit** and **Delete** button to change a warning number or to remove a number from the list.

## Command line syntax

```
--no-warnings[=number,...]
```

```
-w[number,...]
```

## Description

With this option you can suppresses all warning messages or specific warning messages.

On the command line this option works as follows:

- If you do not specify this option, all warnings are reported.
- If you specify this option but without numbers, all warnings are suppressed.
- If you specify this option with a number, only the specified warning is suppressed. You can specify the option **--no-warnings=***number* multiple times.

## Example

To suppress warnings 201 and 202, enter:

```
asmcs test.asm --no-warnings=201,202
```

## Related information

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

# Assembler option: --optimize (-O)

## Menu entry

1. Select **Assembler » Optimization**.
2. Select one or more of the following options:
   - Optimize instruction size

## Command line syntax

**`--optimize=`**`*flag*,...`

**`-O`***`flags`*

You can set the following flags:

| | | |
|---|---|---|
| **+/-instr-size** | **s/S** | Optimize instruction size |

Default: **`--optimize=s`**

## Description

With this option you can control the level of optimization. For details about each optimization see Section 2.4, *Assembler Optimizations*.

## Related information

Section 2.4, *Assembler Optimizations*

# Assembler option: --option-file (-f)

## Menu entry

1. Select **Assembler » Miscellaneous**.
2. Add the option **--option-file** to the **Additional options** field.

   *Be aware that the options in the option file are added to the assembler options you have set in the other pages. Only in extraordinary cases you may want to use them in combination.*

## Command line syntax

```
--option-file=file,...
```

```
-f file,...
```

## Description

This option is primarily intended for command line use. Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the assembler.

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

Option files can also be generated on the fly, for example by the make utility. You can specify the option **--option-file** multiple times.

### Format of an option file

- Multiple arguments on one line in the option file are allowed.
- To include whitespace in an argument, surround the argument with single or double quotes.
- If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

  ```
  "This has a single quote ' embedded"

  'This has a double quote " embedded'

  'This has a double quote " and a single quote '"' embedded"
  ```

- When a text line reaches its length limit, use a **** to continue the line. Whitespace between quotes is preserved.

  ```
  "This is a continuation \
  line"

           -> "This is a continuation line"
  ```

- It is possible to nest command line files up to 25 levels.

## Example

Suppose the file `myoptions` contains the following lines:

```
--debug=+asm,-local
test.asm
```

Specify the option file to the assembler:

```
asmcs --option-file=myoptions
```

This is equivalent to the following command line:

```
asmcs --debug=+asm,-local test.asm
```

## Related information

-

# Assembler option: --output (-o)

## Menu entry

Eclipse names the output file always after the input file.

## Command line syntax

**--output=***file*

**-o** *file*

## Description

With this option you can specify another filename for the output file of the assembler. Without this option, the basename of the assembly source file is used with extension `.o`.

## Example

To create the file `relobj.o` instead of `asm.o`, enter:

```
asmcs --output=relobj.o asm.asm
```

## Related information

-

# Assembler option: --page-length

## Menu entry

1. Select **Assembler » Miscellaneous**.
2. Add the option **--page-length** to the **Additional options** field.

## Command line syntax

```
--page-length=number
```

Default: 72

## Description

If you generate a list file with the assembler option **--list-file**, this option sets the number of lines in a page in the list file. The default is 72, the minimum is 10. As a special case, a page length of 0 turns off page breaks.

## Related information

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

Assembler control **$PAGE**

# Assembler option: --page-width

## Menu entry

1. Select **Assembler » Miscellaneous**.

2. Add the option **--page-width** to the **Additional options** field.

## Command line syntax

```
--page-width=number
```

Default: 132

## Description

If you generate a list file with the assembler option **--list-file**, this option sets the number of columns per line on a page in the list file. The default is 132, the minimum is 40.

## Related information

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

Assembler control **$PAGE**

# Assembler option: --preprocess (-E)

## Menu entry

-

## Command line syntax

```
--preprocess
```

```
-E
```

## Description

With this option the assembler will only preprocess the assembly source file. The assembler sends the preprocessed file to stdout.

## Related information

-

# Assembler option: --preprocessor-type (-m)

## Menu entry

-

## Command line syntax

**`--preprocessor-type=`***`type`*

**`-m`***`type`*

You can set the following preprocessor types:

| | | |
|---|---|---|
| **none** | **n** | No preprocessor |
| **tasking** | **t** | TASKING preprocessor |

Default: **`--preprocessor-type=tasking`**

## Description

With this option you select the preprocessor that the assembler will use. By default, the assembler uses the TASKING preprocessor.

When the assembly source file does not contain any preprocessor symbols, you can specify to the assembler not to use a preprocessor.

## Related information

-

# Assembler option: --section-info (-t)

## Menu entry

1. Select **Assembler » List File**.
2. Enable the option **Generate list file**.
3. Enable the option **List section summary**.

and/or

1. Select **Assembler » Diagnostics**.
2. Enable the option **Display section summary**.

## Command line syntax

**`--section-info`**`[=`*`flag`*`,...]`

**`-t`**`[`*`flags`*`]`

You can set the following flags:

| | | |
|---|---|---|
| **+/-console** | **c/C** | Display section summary on console |
| **+/-list** | **l/L** | List section summary in list file |

Default: **`--section-info=CL`**

Default (without flags): **`--section-info=cl`**

## Description

With this option you tell the assembler to display section information. For each section its memory space, size, total cycle counts and name is listed on stdout and/or in the list file.

The cycle count consists of two parts: the total accumulated count for the section and the total accumulated count for all repeated instructions. In the case of nested loops it is possible that the total supersedes the section total.

## Example

To writes the section information to the list file and also display the section information on stdout, enter:

```
asmcs --list-file --section-info asm.asm
```

## Related information

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

# Assembler option: --symbol-scope (-i)

## Menu entry

1. Select **Assembler » Symbols**.
2. Enable or disable the option **Set default symbol scope to global**.

## Command line syntax

**`--symbol-scope=`***`scope`*

**`-i`***`scope`*

You can set the following scope:

| | | |
|---|---|---|
| **global** | **g** | Default symbol scope is global |
| **local** | **l** | Default symbol scope is local |

Default: **`--symbol-scope=local`**

## Description

With this option you tell the assembler how to treat symbols that you have not specified explicitly as global or local. By default the assembler treats all symbols as local symbols unless you have defined them explicitly as global.

## Related information

Assembler directive **.GLOBAL**

Assembler directive **.LOCAL**

Assembler control **$IDENT**

# Assembler option: --version (-V)

## Menu entry

-

## Command line syntax

```
--version
```

```
-V
```

## Description

Display version information. The assembler ignores all other options or input files.

## Related information

-

## Assembler option: --warnings-as-errors

### Menu entry

1. Select **Global Options**.

2. Enable the option **Treat warnings as errors**.

### Command line syntax

```
--warnings-as-errors[=number,...]
```

### Description

If the assembler encounters an error, it stops assembling. When you use this option without arguments, you tell the assembler to treat all warnings as errors. This means that the exit status of the assembler will be non-zero after one or more assembler warnings. As a consequence, the assembler now also stops after encountering a warning.

You can limit this option to specific warnings by specifying a comma-separated list of warning numbers.

### Related information

Assembler option **--no-warnings** (Suppress some or all warnings)

# 5.2. Linker Options

This section lists all linker options.

## Options in Eclipse versus options on the command line

Most command line options have an equivalent option in Eclipse but some options are only available on the command line. Eclipse invokes the linker via the control program. Therefore, it uses the syntax of the control program to pass options and files to the linker. If there is no equivalent option in Eclipse, you can specify a command line option in Eclipse as follows:

1. From the **Project** menu, select **Properties for**

   *The Properties dialog appears*.

2. In the left pane, expand **C/C++ Build** and select **Settings**.

   *In the right pane the Settings appear.*

3. On the Tool Settings tab, select **Linker » Miscellaneous**.

4. In the **Additional options** field, enter one or more command line options.

   *Because Eclipse uses the control program, Eclipse automatically precedes the option with **-Wl** to pass the option via the control program directly to the linker.*

Be aware that some command line options are not useful in Eclipse or just do not have any effect. For example, the option **--keep-output-files** keeps files after an error occurred. When you specify this option in Eclipse, it will have no effect because Eclipse always removes the output file after an error had occurred.

## Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (**-**) character, long option names always begin with two minus (**--**) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

Options can have flags or suboptions. To switch a flag 'on', use a lowercase letter or a **+***longflag*. To switch a flag off, use an uppercase letter or a **-***longflag*. Separate *longflags* with commas. The following two invocations are equivalent:

```
lmcs -mfkl test.o
lmcs --map-file-format=+files,+link,+locate test.o
```

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

# Linker option: --case-insensitive

## Menu entry

1. Select **Linker » Miscellaneous**.

2. Enable the option **Link case insensitive**.

## Command line syntax

```
--case-insensitive
```

Default: case sensitive

## Description

With this option you tell the linker not to distinguish between uppercase and lowercase characters in symbols. By default the linker considers uppercase and lowercase characters as different characters.

When you have written your own assembly code and specified to assemble it case insensitive, you must also link the `.o` file case insensitive.

## Related information

Assembler option **--case-insensitive**

# Linker option: --chip-output (-c)

## Menu entry

1. Select **Linker » Output Format**.
2. Enable the option **Generate Intel Hex format file** and/or **Generate S-records file**.
3. Enable the option **Create file for each memory chip**.
4. Optionally, specify the **Size of addresses**.

   *Eclipse always uses the project name as the basename for the output file.*

## Command line syntax

```
--chip-output=[basename]:format[:addr_size],...
```

```
-c[basename]:format[:addr_size],...
```

You can specify the following formats:

| | |
|---|---|
| **IHEX** | Intel Hex |
| **SREC** | Motorola S-records |

The *addr_size* specifies the size of the addresses in bytes (record length). For Intel Hex you can use the values **1**, **2** or **4** bytes (default). For Motorola-S you can specify: **2** (S1 records), **3** (S2 records) or **4** bytes (S3 records, default).

## Description

With this option you specify the Intel Hex or Motorola S-record output format for loading into a PROM-programmer. The linker generates a file for each ROM memory defined in the LSL file, where sections are located:

```
memory memname
{  type=rom;  }
```

The name of the file is the name of the Eclipse project or, on the command line, the name of the memory device that was emitted with extension `.hex` or `.sre`. Optionally, you can specify a *basename* which prepends the generated file name.

The linker always outputs a debugging file in ELF/DWARF format and optionally an absolute object file in Intel Hex-format and/or Motorola S-record format.

## Example

To generate Intel Hex output files for each defined memory, enter the following on the command line:

```
lmcs --chip-output=myfile:IHEX test1.o
```

In this case, this generates the file `myfile_`*memname*`.hex`.

## Related information

Linker option **--output** (Output file)

# Linker option: --core (-C)

## Menu entry

1. Expand **C/C++ Build** and select **Processor**.
2. From the **Processor Selection** list, select a processor or select **User defined TriCore ...**.
3. From the **Multi-core configuration** list, select an MCS core.

## Command line syntax

**--core=***MCS-core*

**-C***MCS-core*

You can specify the following MCS cores:

| | |
|---|---|
| **mpe:mcs00** | MCS core 0 |
| **mpe:mcs01** | MCS core 1 |
| **mpe:mcs02** | MCS core 2 |
| **mpe:mcs03** | MCS core 3 |

Default: **mpe:mcs00**

## Description

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

In a multi-task setting, use this option to tell the linker to use a specific core for a specific task. Only one task can be assigned to a certain core. Assigning multiple tasks to a single core requires some form of kernel functionality.

The core is specified as `mpe:mcs0{0123}`. For example, the file `tc27x.lsl` in the `include.lsl` directory, contains a description of derivative `tc27x` and the supported MCS cores. `mpe` is the multi-processor environment as specified in the LSL file.

## Example

To link objects for the MCS core `mpe:mcs01`, enter:

```
lcms  -o test.elf -dtc27x.lsl --non-romable
      --user-provided-initialization-code -D__LINKONLY__
      -DCSA=0 --core=mpe:mcs01 --map-file test.o
```

## Related information

Control program option **--lsl-core** (Specify LSL core)

# Linker option: --define (-D)

## Menu entry

1. Select **Linker » Script File**.

   *The Defined symbols box shows the symbols that are currently defined.*

2. To define a new symbol, click on the **Add** button in the **Defined symbols** box.

3. Type the symbol definition (for example, `demo=1`)

Use the **Edit** and **Delete** button to change a macro definition or to remove a macro from the list.

## Command line syntax

**`--define=`***`macro_name`*`[`**`=`***`macro_definition`*`]`

**`-D`***`macro_name`*`[`**`=`***`macro_definition`*`]`

## Description

With this option you can define a macro and specify it to the linker LSL file preprocessor. If you only specify a macro name (no macro definition), the macro expands as '1'.

You can specify as many macros as you like; just use the option **--define** (**-D**) multiple times. If the command line exceeds the limit of the operating system, you can define the macros in an option file which you then must specify to the linker with the option **--option-file** (**-f**) *file*.

The definition can be tested by the preprocessor with `#if`, `#ifdef` and `#ifndef`, for conditional locating.

## Example

To define the RESET vector, which is used in the linker script file `tc27x.lsl`, enter:

```
lmcs test.o -otest.elf --lsl-file=tc27x.lsl --define=RESET=0xa0000020
```

## Related information

Linker option **--option-file** (Specify an option file)

# Linker option: --diag

## Menu entry

1. From the **Window** menu, select **Show View » Other » TASKING » Problems**.

   *The Problems view is added to the current perspective.*

2. In the Problems view right-click on a message.

   *A popup menu appears.*

3. Select **Detailed Diagnostics Info**.

   *A dialog box appears with additional information.*

## Command line syntax

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

You can set the following output formats:

| | |
|---|---|
| **html** | HTML output. |
| **rtf** | Rich Text Format. |
| **text** | ASCII text. |

Default format: text

## Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to stdout (normally your screen) and in the format you specify. You can specify the following formats: **html**, **rtf** or **text** (default). To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas.

With this option the linker does not link/locate any files.

## Example

To display an explanation of message number 106, enter:

```
lmcs --diag=106
```

This results in the following message and explanation:

```
E106: unresolved external: <message>

The linker could not resolve all external symbols.
```

```
This is an error when the incremental linking option is disabled.
The <message> indicates the symbol that is unresolved.
```

To write an explanation of all errors and warnings in HTML format to file `lkerrors.html`, use redirection and enter:

```
lmcs --diag=html:all > lkerrors.html
```

## Related information

Section 3.9, *Linker Error Messages*

# Linker option: --error-file

## Menu entry

-

## Command line syntax

**`--error-file`**`[=`*`file`*`]`

## Description

With this option the linker redirects error messages to a file. If you do not specify a filename, the error file is `lmcs.elk`.

## Example

To write errors to `errors.elk` instead of `stderr`, enter:

```
lmcs --error-file=errors.elk test.o
```

## Related information

Section 3.9, *Linker Error Messages*

## Linker option: --error-limit

### Menu entry

1. Select **Linker » Diagnostics**.

2. Enter a value in the **Maximum number of emitted errors** field.

### Command line syntax

**`--error-limit=`***`number`*

Default: 42

### Description

With this option you tell the linker to only emit the specified maximum number of errors. When 0 (null) is specified, the linker emits all errors. Without this option the maximum number of errors is 42.

### Related information

Section 3.9, *Linker Error Messages*

# Linker option: --extern (-e)

## Menu entry

-

## Command line syntax

```
--extern=symbol,...
```

```
-esymbol,...
```

## Description

With this option you force the linker to consider the given symbol as an undefined reference. The linker tries to resolve this symbol, either the symbol is defined in an object file or the linker extracts the corresponding symbol definition from a library.

This option is, for example, useful if the startup code is part of a library. Because your own application does not refer to the startup code, you can force the startup code to be extracted by specifying the symbol _START as an unresolved external.

## Example

Consider the following invocation:

```
lmcs 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`.

```
lmcs --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

-

# Linker option: --first-library-first

## Menu entry

-

## Command line syntax

```
--first-library-first
```

## Description

When the linker processes a library it searches for symbols that are referenced by the objects and libraries processed so far. If the library contains a definition for an unresolved reference the linker extracts the object that contains the definition from the library.

By default the linker processes object files and libraries in the order in which they appear on the command line. If you specify the option **--first-library-first** the linker always tries to take the symbol definition from the library that appears first on the command line before scanning subsequent libraries.

This is for example useful when you are working with a newer version of a library that partially overlaps the older version. Because they do not contain exactly the same functions, you have to link them both. However, when a function is present in both libraries, you may want the linker to extract the most recent function.

## Example

Consider the following example:

```
lmcs --first-library-first a.a test.o b.a
```

If the file `test.o` calls a function which is both present in `a.a` and `b.a`, normally the function in `b.a` would be extracted. With this option the linker first tries to extract the symbol from the first library `a.a`.

> Note that routines in `b.a` that call other routines that are present in both `a.a` and `b.a` are now also resolved from `a.a`.

## Related information

Linker option **--no-rescan** (Rescan libraries to solve unresolved externals)

# Linker option: --help (-?)

## Menu entry

-

## Command line syntax

**--help**[**=***item*]

**-?**

You can specify the following arguments:

| | |
|---|---|
| **options** | Show extended option descriptions |

## Description

Displays an overview of all command line options. When you specify the argument **options** you can list detailed option descriptions.

## Example

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

```
lmcs -?
lmcs --help
lmcs
```

To see a detailed description of the available options, enter:

```
lmcs --help=options
```

## Related information

-

# Linker option: --hex-format

## Menu entry

1. Select **Linker » Miscellaneous**.
2. Add the option **--hex-format** to the **Additional options** field.

## Command line syntax

```
--hex-format=flag,...
```

You can set the following flag:

| | | |
|---|---|---|
| **+/-start-address** | **s/S** | Emit start address record |

Default: `--hex-format=s`

## Description

With this option you can specify to emit or omit the start address record from the hex file.

## Related information

Linker option **--output** (Output file)

# Linker option: --hex-record-size

## Menu entry

1. Select **Linker » Miscellaneous**.
2. Add the option **--hex-record-size** to the **Additional options** field.

## Command line syntax

```
--hex-record-size=size
```

Default: 32

## Description

With this option you can set the size (width) of the Intel Hex data records.

## Related information

Linker option **--output** (Output file)

## Linker option: --import-object

### Menu entry

1. Select **Linker » Data Objects**.

   *The Data objects box shows the list of object files that are imported.*

2. To add a data object, click on the **Add** button in the **Data objects** box.

3. Type or select a binary file (including its path).

> Use the **Edit** and **Delete** button to change a filename or to remove a data object from the list.

### Command line syntax

```
--import-object=file,...
```

### Description

With this option the linker imports a binary *file* containing raw data and places it in a section. The section name is derived from the filename, in which dots are replaced by an underscore. So, when importing a file called `my.jpg`, a section with the name `my_jpg` is created. In your application you can refer to the created section by using linker labels.

### Related information

Section 3.4, *Importing Binary Files*

# Linker option: --include-directory (-I)

## Menu entry

-

## Command line syntax

**`--include-directory=`**`path,...`

**`-I`**`path,...`

## Description

With this option you can specify the path where your LSL include files are located. A relative path will be relative to the current directory.

The order in which the linker searches for LSL include files is:

1. The pathname in the LSL file and the directory where the LSL file is located (only for #include files that are enclosed in "")
2. The path that is specified with this option.
3. The default directory `$(PRODDIR)\include.lsl`.

## Example

Suppose that your linker script file `mylsl.lsl` contains the following line:

```
#include "myinc.inc"
```

You can call the linker as follows:

```
lmcs --include-directory=c:\proj\include --lsl-file=mylsl.lsl test.o
```

First the linker looks for the file `myinc.inc` in the directory where `mylsl.lsl` is located. If it does not find the file, it looks in the directory `c:\proj\include` (this option). Finally it looks in the directory `$(PRODDIR)\include.lsl`.

## Related information

Linker option **--lsl-file** (Specify linker script file)

## Linker option: --incremental (-r)

### Menu entry

-

### Command line syntax

**`--incremental`**

**`-r`**

### Description

Normally the linker links and locates the specified object files. With this option you tell the linker only to link the specified files. The linker creates a linker output file `.out`. You then can link this file again with other object files until you have reached the final linker output file that is ready for locating.

In the last pass, you call the linker without this option with the final linker output file `.out`. The linker will now locate the file.

### Example

In this example, the files `test1.o`, `test2.o` and `test3.o` are incrementally linked:

1. `lmcs --incremental test1.o test2.o --output=test.out`

   *test1.o and test2.o are linked*

2. `lmcs --incremental test3.o test.out`

   *test3.o and test.out are linked, task1.out is created*

3. `lmcs task1.out`

   *task1.out is located*

### Related information

Section 3.3, *Incremental Linking*

# Linker option: --keep-output-files (-k)

## Menu entry

Eclipse *always* removes the output files when errors occurred.

## Command line syntax

```
--keep-output-files
```

```
-k
```

## Description

If an error occurs during linking, the resulting output file may be incomplete or incorrect. With this option you keep the generated output files when an error occurs.

By default the linker removes the generated output file when an error occurs. This is useful when you use the make utility. If the erroneous files are not removed, the make utility may process corrupt files on a subsequent invocation.

Use this option when you still want to use the generated file. For example when you know that a particular error does not result in a corrupt object file, or when you want to inspect the output file, or send it to Altium support.

## Related information

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

# Linker option: --library (-l)

## Menu entry

1. Select **Linker » Libraries**.

   *The Libraries box shows the list of libraries that are linked with the project.*

2. To add a library, click on the **Add** button in the **Libraries** box.

3. Type or select a library (including its path).

4. Optionally, disable the option **Link default libraries**.

> Use the **Edit** and **Delete** button to change a library name or to remove a library from the list.

## Command line syntax

**`--library=`***`name`*

**`-l`***`name`*

## Description

With this option you tell the linker to use system library `lib`*`name`*`.a`, where *name* is a string. The linker first searches for system libraries in any directories specified with **--library-directory**, then in the directories specified with the environment variable `LIBTC1V1_6_X`, unless you used the option **--ignore-default-library-path**.

## Example

To search in the system library `libc.a`:

```
lmcs test.o mylib.a --library=c
```

The linker links the file `test.o` and first looks in library `mylib.a` (in the current directory only), then in the system library `libc.a` to resolve unresolved symbols.

## Related information

Linker option **--library-directory** (Additional search path for system libraries)

-

# Linker option: --library-directory (-L) / --ignore-default-library-path

## Menu entry

1. Select **Linker » Libraries**.

   *The Library search path box shows the directories that are added to the search path for library files.*

2. To define a new directory for the search path, click on the **Add** button in the **Library search path** box.

3. Type or select a path.

Use the **Edit** and **Delete** button to change a path or to remove a path from the list.

## Command line syntax

```
--library-directory=path,...
-Lpath,...
```

```
--ignore-default-library-path
-L
```

## Description

With this option you can specify the path(s) where your system libraries, specified with the option **--library** (**-l**), are located. If you want to specify multiple paths, use the option **--library-directory** for each separate path.

The default path is `$(PRODDIR)\lib`.

If you specify only **-L** (without a pathname) or the long option **--ignore-default-library-path**, the linker will not search the default path and also not in the paths specified in the environment variables `LIBTC1V1_6_X`. So, the linker ignores steps 2 and 3 as listed below.

The priority order in which the linker searches for system libraries specified with the option **--library** (**-l**) is:

1. The path that is specified with the option **--library-directory**.

2. The path that is specified in the environment variables `LIBTC1V1_6_X`.

3. The default directory `$(PRODDIR)\lib`.

## Example

Suppose you call the linker as follows:

```
lmcs test.o --library-directory=c:\mylibs --library=c
```

First the linker looks in the directory `c:\mylibs` for library `libc.a` (this option). If it does not find the requested libraries, it looks in the directory that is set with the environment variables `LIBTC1V1_6_X`. Then the linker looks in the default directory `$(PRODDIR)\lib` for libraries.

## Related information

Linker option **--library** (Link system library)

-

# Linker option: --link-only

## Menu entry

-

## Command line syntax

```
--link-only
```

## Description

With this option you suppress the locating phase. The linker stops after linking and informs you about unresolved references.

## Related information

Control program option **--create=relocatable** (**-cl**) (Stop after linking)

## Linker option: --lsl-check

### Menu entry

-

### Command line syntax

```
--lsl-check
```

### Description

With this option the linker just checks the syntax of the LSL file(s) and exits. No linking or locating is performed. Use the option **--lsl-file** to specify the name of the Linker Script File you want to test.

### Related information

Linker option **--lsl-file** (Linker script file)

Linker option **--lsl-dump** (Dump LSL info)

Section 3.6, *Controlling the Linker with a Script*

# Linker option: --lsl-dump

## Menu entry

-

## Command line syntax

**`--lsl-dump`**`[=`*`file`*`]`

## Description

With this option you tell the linker to dump the LSL part of the map file in a separate file, independent of the option **--map-file** (generate map file). If you do not specify a filename, the file `lmcs.ldf` is used.

## Related information

Linker option **--map-file-format** (Map file formatting)

# Linker option: --lsl-file (-d)

## Menu entry

An LSL file can be generated when you create your TriCore project in Eclipse:

1. From the **File** menu, select **File » New » TASKING TriCore C/C++ Project**.

   *The New C/C++ Project wizard appears.*

2. Fill in the project settings in each dialog and click **Next >** until the **TriCore Project Settings** appear.

3. Enable the option**Add linker script file to the project** and click **Finish**.

   *Eclipse creates your project and the file project*`.lsl` *in the project directory.*

The LSL file can be specified in the Properties dialog:

1. Select **Linker » Script File**.

2. Specify a LSL file in the **Linker script file (.lsl)** field.

## Command line syntax

**`--lsl-file=`**`file`

**`-d`**`file`

## Description

A linker script file contains vital information about the core for the locating phase of the linker. A linker script file is coded in LSL and contains the following types of information:

- the architecture definition describes the core's hardware architecture.

- the memory definition describes the physical memory available in the system.

- the section layout definition describes how to locate sections in memory.

With this option you specify a linker script file to the linker. If you do not specify this option, the linker uses a default script file. You can specify the existing file *target*`.lsl` or the name of a manually written linker script file. You can use this option multiple times. The linker processes the LSL files in the order in which they appear on the command line.

## Related information

Linker option **--lsl-check** (Check LSL file(s) and exit)

Section 3.6, *Controlling the Linker with a Script*

# Linker option: --map-file (-M)

## Menu entry

1. Select **Linker » Map File**.
2. Enable the option **Generate XML map file format (.mapxml) for map file viewer**.
3. (Optional) Enable the option **Generate map file**.
4. Enable or disable the types of information to be included.

## Command line syntax

**`--map-file`**`[=`*`file`*`][`**`:XML`**`]`

**`-M`**`[`*`file`*`][`**`:XML`**`]`

Default (Eclipse): XML map file is generated

Default (linker): no map file is generated

## Description

With this option you tell the linker to generate a linker map file. If you do not specify a filename and you specified the option **--output**, the linker uses the same basename as the output file with the extension `.map`. If you did not specify the option **--output**, the linker uses the file `task1.map`. Eclipse names the `.map` file after the project.

A linker map file is a text file that shows how the linker has mapped the sections and symbols from the various object files (`.o`) to the linked object file. A locate part shows the absolute position of each section. External symbols are listed per space with their absolute address, both sorted on symbol and sorted on address.

## Related information

Linker option **--map-file-format** (Format map file)

Section 6.2, *Linker Map File Format*

# Linker option: --map-file-format (-m)

## Menu entry

1. Select **Linker » Map File**.
2. Enable the option **Generate XML map file format (.mapxml) for map file viewer**.
3. (Optional) Enable the option **Generate map file**.
4. Enable or disable the types of information to be included.

## Command line syntax

```
--map-file-format=flag,...
```

```
-mflags
```

You can set the following flags:

| | | |
|---|---|---|
| **+/-callgraph** | **c/C** | Include call graph information |
| **+/-removed** | **d/D** | Include information on removed sections |
| **+/-files** | **f/F** | Include processed files information |
| **+/-invocation** | **i/I** | Include information on invocation and tools |
| **+/-link** | **k/K** | Include link result information |
| **+/-locate** | **l/L** | Include locate result information |
| **+/-memory** | **m/M** | Include memory usage information |
| **+/-nonalloc** | **n/N** | Include information of non-alloc sections |
| **+/-overlay** | **o/O** | Include overlay information |
| **+/-statics** | **q/Q** | Include module local symbols information |
| **+/-crossref** | **r/R** | Include cross references information |
| **+/-lsl** | **s/S** | Include processor and memory information |
| **+/-rules** | **u/U** | Include locate rules |

Use the following options for predefined sets of flags:

| | | |
|---|---|---|
| **--map-file-format=0** | **-m0** | Link information<br>Alias for **-mcDfikLMNoQrSU** |
| **--map-file-format=1** | **-m1** | Locate information<br>Alias for **-mCDfiKlMNoQRSU** |
| **--map-file-format=2** | **-m2** | Most information<br>Alias for **-mcdfiklmNoQrSu** |

Default: **`--map-file-format=2`**

## Description

With this option you specify which information you want to include in the map file.

On the command line you must use this option in combination with the option **--map-file** (**-M**).

## Related information

Linker option **--map-file** (Generate map file)

Section 6.2, *Linker Map File Format*

# Linker option: --new-task

## Menu entry

-

## Command line syntax

```
--new-task
```

## Description

With this option the linker creates an additional task. Any options that follow only apply to the new task.

The linker processes options on the command line from left to right. To know whether a certain option belongs to a different task it uses this option. This implies that all options for a given task must be fully specified before moving on to the next.

## Related information

-

# Linker option: --non-romable

## Menu entry

-

## Command line syntax

```
--non-romable
```

## Description

With this option you tell the linker that the application must not be located in ROM. The linker will locate all ROM sections, including a copy table if present, in RAM. When the application is started, the data sections are re-initialized and the BSS sections are cleared as usual.

This option is, for example, useful when you want to test the application in RAM before you put the final application in ROM. This saves you the time of flashing the application in ROM over and over again.

## Related information

-

## Linker option: --no-rescan

### Menu entry

1. Select **Linker » Libraries**.
2. Disable the option **Rescan libraries to solve unresolved externals**.

### Command line syntax

```
--no-rescan
```

### Description

When the linker processes a library it searches for symbol definitions that are referenced by the objects and libraries processed so far. If the library contains a definition for an unresolved reference the linker extracts the object that contains the definition from the library. The linker processes object files and libraries in the order in which they appear on the command line.

When all objects and libraries are processed the linker checks if there are unresolved symbols left. If so, the default behavior of the linker is to rescan all libraries in the order given at the command line. The linker stops rescanning the libraries when all symbols are resolved, or when the linker could not resolve any symbol(s) during the rescan of all libraries. Notice that resolving one symbol may introduce new unresolved symbols.

With this option, you tell the linker to scan the object files and libraries only once. When the linker has not resolved all symbols after the first scan, it reports which symbols are still unresolved. This option is useful if you are building your own libraries. The libraries are most efficiently organized if the linker needs only one pass to resolve all symbols.

### Related information

Linker option **--first-library-first** (Scan libraries in given order)

# Linker option: --no-rom-copy (-N)

## Menu entry

-

## Command line syntax

**`--no-rom-copy`**

**`-N`**

## Description

With this option the linker will not generate a ROM copy for data sections. A copy table is generated and contains entries to clear BSS sections. However, no entries to copy data sections from ROM to RAM are placed in the copy table.

The data sections are initialized when the application is downloaded. The data sections are not re-initialized when the application is restarted.

## Related information

-

# Linker option: --no-warnings (-w)

## Menu entry

1. Select **Linker » Diagnostics**.

   *The Suppress warnings box shows the warnings that are currently suppressed.*

2. To suppress a warning, click on the **Add** button in the **Suppress warnings** box.

3. Enter the numbers, separated by commas, of the warnings you want to suppress (for example `135,136`). Or you can use the **Add** button multiple times.

4. To suppress all warnings, enable the option **Suppress all warnings**.

Use the **Edit** and **Delete** button to change a warning number or to remove a number from the list.

## Command line syntax

**`--no-warnings`**`[=`*`number`*`,...]`

**`-w`**`[`*`number`*`,...]`

## Description

With this option you can suppresses all warning messages or specific warning messages.

On the command line this option works as follows:

- If you do not specify this option, all warnings are reported.

- If you specify this option but without numbers, all warnings are suppressed.

- If you specify this option with a number, only the specified warning is suppressed. You can specify the option **--no-warnings=***number* multiple times.

## Example

To suppress warnings 135 and 136, enter:

```
lmcs --no-warnings=135,136 test.o
```

## Related information

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

# Linker option: --optimize (-O)

## Menu entry

1. Select **Linker » Optimization**.
2. Select one or more of the following options:
   - Delete unreferenced sections
   - Use a 'first-fit decreasing' algorithm
   - Compress copy table
   - Delete duplicate code
   - Delete duplicate data

## Command line syntax

**`--optimize=`***`flag`*`,...`

**`-O`***`flags`*

You can set the following flags:

| | | |
|---|---|---|
| **+/-delete-unreferenced-sections** | **c/C** | Delete unreferenced sections from the output file |
| **+/-first-fit-decreasing** | **l/L** | Use a 'first-fit decreasing' algorithm to locate unrestricted sections in memory |
| **+/-copytable-compression** | **t/T** | Emit smart restrictions to reduce copy table size |
| **+/-delete-duplicate-code** | **x/X** | Delete duplicate code sections from the output file |
| **+/-delete-duplicate-data** | **y/Y** | Delete duplicate constant data from the output file |

Use the following options for predefined sets of flags:

| | | |
|---|---|---|
| **--optimize=0** | **-O0** | No optimization<br>Alias for **-OCLTXY** |
| **--optimize=1** | **-O1** | Default optimization<br>Alias for **-OcLtxy** |
| **--optimize=2** | **-O2** | All optimizations<br>Alias for **-Ocltxy** |

Default: **`--optimize=1`**

## Description

With this option you can control the level of optimization.

## Related information

For details about each optimization see Section 3.5, *Linker Optimizations*.

## Linker option: --option-file (-f)

### Menu entry

1. Select **Linker » Miscellaneous**.
2. Add the option **--option-file** to the **Additional options** field.

   *Be aware that the options in the option file are added to the linker options you have set in the other pages. Only in extraordinary cases you may want to use them in combination.*

### Command line syntax

**--option-file=***file*,...

**-f** *file*,...

### Description

This option is primarily intended for command line use. Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the linker.

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

Option files can also be generated on the fly, for example by the make utility. You can specify the option **--option-file** multiple times.

#### Format of an option file

- Multiple arguments on one line in the option file are allowed.
- To include whitespace in an argument, surround the argument with single or double quotes.
- If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

  ```
  "This has a single quote ' embedded"

  'This has a double quote " embedded'

  'This has a double quote " and a single quote '"' embedded"
  ```

- When a text line reaches its length limit, use a **\** to continue the line. Whitespace between quotes is preserved.

  ```
  "This is a continuation \
  line"

           -> "This is a continuation line"
  ```

- It is possible to nest command line files up to 25 levels.

## Example

Suppose the file `myoptions` contains the following lines:

```
--map-file=my.map               (generate a map file)
test.o                          (input file)
--library-directory=c:\mylibs   (additional search path for system libraries)
```

Specify the option file to the linker:

```
lmcs --option-file=myoptions
```

This is equivalent to the following command line:

```
lmcs --map-file=my.map test.o --library-directory=c:\mylibs
```

## Related information

-

# Linker option: --output (-o)

## Menu entry

1. Select **Linker » Output Format**.
2. Enable one or more output formats.

   *For some output formats you can specify a number of suboptions.*

   *Eclipse always uses the project name as the basename for the output file.*

## Command line syntax

```
--output=[filename][:format[:addr_size][,space_name]]...
```

```
-o[filename][:format[:addr_size][,space_name]]...
```

You can specify the following formats:

| | |
|---|---|
| **ELF** | ELF/DWARF |
| **IHEX** | Intel Hex |
| **SREC** | Motorola S-records |

## Description

By default, the linker generates an output file in ELF/DWARF format, with the name `task1.elf`.

With this option you can specify an alternative *filename*, and an alternative output *format*. The default output format is the format of the first input file.

You can use the **--output** option multiple times. This is useful to generate multiple output formats. With the first occurrence of the **--output** option you specify the basename (the filename without extension), which is used for subsequent **--output** options with no filename specified. If you do not specify a filename, or you do not specify the **--output** option at all, the linker uses the default basename `task`*n*.

### IHEX and SREC formats

If you specify the Intel Hex format or the Motorola S-records format, you can use the argument *addr_size* to specify the size of addresses in bytes (record length). For Intel Hex you can use the values: 1, 2, and 4 (default). For Motorola S-records you can specify: 2 (S1 records), 3 (S2 records, default) or 4 bytes (S3 records).

With the argument *space_name* you can specify the name of the address space. The name of the output file will be filename with the extension `.hex` or `.sre` and contains the code and data allocated in the specified space. If they exist, any other address spaces are also emitted whereas their output files are named *filename_spacename* with the extension `.hex` or `.sre`.

If you do not specify *space_name*, or you specify a non-existing space, the default address space is filled in.

Use option **--chip-output** (**-c**) to create Intel Hex or Motorola S-record output files for each chip defined in the LSL file (suitable for loading into a PROM-programmer).

## Example

To create the output file `myfile.hex` of the address space named `linear`, enter:

```
lmcs test.o --output=myfile.hex:IHEX:2,linear
```

If they exist, any other address spaces are emitted as well and are named `myfile_`*spacename*`.hex`.

## Related information

Linker option **--chip-output** (Generate an output file for each chip)

Linker option **--hex-format** (Specify Hex file format settings)

## Linker option: --strip-debug (-S)

### Menu entry

1. Select **Linker » Miscellaneous**.

2. Enable the option **Strip symbolic debug information**.

### Command line syntax

```
--strip-debug
```

```
-S
```

### Description

With this option you specify not to include symbolic debug information in the resulting output file.

### Related information

-

# Linker option: --user-provided-initialization-code (-i)

## Menu entry

1. Select **Linker » Miscellaneous**.
2. Enable the option **Do not use standard copy table for initialization**.

## Command line syntax

```
--user-provided-initialization-code

-i
```

## Description

It is possible to use your own initialization code, for example, to save ROM space. With this option you tell the linker *not* to generate a copy table for initialize/clear sections. Use linker labels in your source code to access the positions of the sections when located.

If the linker detects references to the TASKING initialization code, an error is emitted: it is either the TASKING initialization routine or your own, not both.

Note that the options **--no-rom-copy** and **--non-romable**, may vary independently. The 'copytable-compression' optimization (**--optimize=t**) is automatically disabled when you enable this option.

## Related information

Linker option **--no-rom-copy** (Do not generate ROM copy)

Linker option **--non-romable** (Application is not romable)

Linker option **--optimize** (Specify optimization)

## Linker option: --verbose (-v)

### Menu entry

-

### Command line syntax

**`--verbose`**

**`-v`**

### Description

With this option you put the linker in *verbose* mode. The linker prints the link phases while it processes the files. The linker prints one entry for each action it executes for a task. When you use this option twice (**`-vv`**) you put the linker in *extra verbose* mode. In this mode the linker also prints the filenames and it shows which objects are extracted from libraries and it shows verbose information that would normally be hidden when you use the normal verbose mode or when you run without verbose. With this option you can monitor the current status of the linker.

### Related information

-

# Linker option: --version (-V)

## Menu entry

-

## Command line syntax

```
--version
```

```
-V
```

## Description

Display version information. The linker ignores all other options or input files.

## Related information

-

# Linker option: --warnings-as-errors

## Menu entry

1. Select **Global Options**.
2. Enable the option **Treat warnings as errors**.

## Command line syntax

```
--warnings-as-errors[=number,...]
```

## Description

When the linker detects an error or warning, it tries to continue the link process and reports other errors and warnings. When you use this option without arguments, you tell the linker to treat all warnings as errors. This means that the exit status of the linker will be non-zero after the detection of one or more linker warnings. As a consequence, the linker will not produce any output files.

You can also limit this option to specific warnings by specifying a comma-separated list of warning numbers.

## Related information

Linker option **--no-warnings** (Suppress some or all warnings)

## 5.3. Control Program Options

The control program **ccmcs** facilitates the invocation of the various components of the MCS toolset from a single command line.

### Options in Eclipse versus options on the command line

Eclipse invokes the assembler and linker via the control program. Therefore, it uses the syntax of the control program to pass options and files to the tools. The control program processes command line options either by itself, or, when the option is unknown to the control program, it looks whether it can pass the option to one of the other tools. However, for directly passing an option to the assembler or linker, it is recommended to use the control program options **--pass-assembler**, **--pass-linker**.

See the previous sections for details on the options of the tools.

### Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (**-**) character, long option names always begin with two minus (**--**) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

Options can have flags or suboptions. To switch a flag 'on', use a lowercase letter or a **+***longflag*. To switch a flag off, use an uppercase letter or a **-***longflag*. Separate *longflags* with commas. The following two invocations are equivalent:

```
ccmcs -Wa-gs test.asm
ccmcs --pass-assembler=--debug-info=+smart test.asm
```

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

# Control program option: --case-insensitive

## Menu entry

1. Select **Assembler » Symbols**.
2. Enable the option **Case insensitive identifiers**.

## Command line syntax

**`--case-insensitive`**

Default: case sensitive

## Description

With this option you tell the assembler not to distinguish between uppercase and lowercase characters. By default the assembler considers uppercase and lowercase characters as different characters.

## Example

When assembling case insensitive, the label `LabelName` is the same label as `labelname`.

```
ccmcs --case-insensitive test.asm
```

## Related information

Assembler option **--case-insensitive**

Assembler control **$CASE**

# Control program option: --check

## Menu entry

-

## Command line syntax

```
--check
```

## Description

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

The assembler reports any warnings and/or errors.

This option is available on the command line only.

## Related information

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

# Control program option: --cpu (-C)

## Menu entry

1. Expand **C/C++ Build** and select **Processor**.
2. From the **Processor Selection** list, select a processor or select **User defined TriCore ...**.

## Command line syntax

**--cpu=***cpu*

**-C***cpu*

## Description

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

The standard list of supported processors is defined in the file `processors.xml`. This file defines for each processor its full name (for example, TC27X), the base CPU name (for example, tc27x) and the core settings (for example, mcs).

The control program passes the options to the underlaying tools. For example, **--cpu=tc27x** to the assembler, or **-dtc27x.lsl --core=mpe:mcs00** to the linker.

## Example

To generate the file `test.elf` for the TC27X processor, enter:

```
ccmcs --cpu=tc27x test.asm
```

## Related information

Control program option **--cpu-list** (Show list of processors)

Control program option **--lsl-core** (Specify LSL core)

Control program option **--processors** (Read additional processor definitions)

## Control program option: --cpu-list

### Menu entry

-

### Command line syntax

**`--cpu-list`**`[=`*`pattern`*`]`

### Description

With this option the control program shows a list of supported processors as defined in the file `processors.xml`. This can be useful when you want to select a processor name or id for the **--cpu** option.

The *pattern* works similar to the UNIX **grep** utility. You can use it to limit the output list.

### Example

To show a list of all processors, enter:

```
ccmcs --cpu-list
```

To show all processors of the mcs core, enter:

```
ccmcs --cpu-list=mcs

--- ~/cmcs/etc/processors.xml ---
    id            name                         CPU           core
    userdef16x    User defined TriCore 1.6.x   userdef16x    mcs
    tc2d5t        TC2D5T                       tc2d5t        mcs
    tc27x         TC27X                        tc27x         mcs
```

### Related information

Control program option **--cpu** (Select processor)

# Control program option: --create (-c)

## Menu entry

-

## Command line syntax

**`--create`**`[=`*`stage`*`]`

**`-c`**`[`*`stage`*`]`

You can specify the following stages:

| | | |
|---|---|---|
| **relocatable** | **l** | Stop after the files are linked to a linker object file (`.out`) |
| **object** | **o** | Stop after the files are assembled to objects (`.o`) |

Default (without flags): **`--create=object`**

## Description

Normally the control program generates an absolute object file of the specified output format from the file you supplied as input. With this option you tell the control program to stop after a certain number of phases.

## Example

To generate the object file `test.o`:

```
ccmcs --create test.asm
```

The control program stops after the file is assembled. It does not link nor locate the generated output.

## Related information

Linker option **--link-only** (Link only, no locating)

# Control program option: --debug-info (-g)

## Menu entry

1. Select **Assembler » Symbols**.
2. Select an option from the **Generate symbolic debug** list.

## Command line syntax

```
--debug-info
```

```
-g
```

## Description

With this option you tell the control program to include debug information in the generated object file.

The control program calls the assembler with **--debug-info=+local,+smart** (**-gls**).

## Related information

Assembler option **--debug-info** (Generate symbolic debug information)

# Control program option: --define (-D)

## Menu entry

1. Select **Assembler » Preprocessing**.

   *The Defined symbols box right-below shows the symbols that are currently defined.*

2. To define a new symbol, click on the **Add** button in the **Defined symbols** box.

3. Type the symbol definition (for example, `demo=1`)

> Use the **Edit** and **Delete** button to change a macro definition or to remove a macro from the list.

## Command line syntax

**`--define=`***`macro_name`*`[`**`=`***`macro_definition`*`]`

**`-D`***`macro_name`*`[`**`=`***`macro_definition`*`]`

## Description

With this option you can define a macro and specify it to the preprocessor. If you only specify a macro name (no macro definition), the macro expands as '1'.

You can specify as many macros as you like. Simply use the **Add** button to add new macro definitions.

On the command line, use the option **--define** (**-D**) multiple times. If the command line exceeds the limit of the operating system, you can define the macros in an option file which you then must specify to the assembler with the option **--option-file** (**-f**) *file*.

Defining macros with this option (instead of in the assembly source) is, for example, useful to assembly conditional assembly source as shown in the example below.

The control program passes the option **--define** (**-D**) to the assembler.

## Example

Consider the following program with conditional code to assemble a demo program and a real program:

```
.SDECL ".mcstext.code",code
.SECT  ".mcstext.code"
.IF DEMO == 1
    JMP demo_part   ; assemble for the demo program
.ELSE
    JMP real_part   ; assemble for the real program
.ENDIF
```

You can now use a macro definition to set the DEMO flag:

```
ccmcs --define=DEMO test.asm
ccmcs --define=DEMO=1 test.asm
```

Note that both invocations have the same effect.

The next example shows how to define a macro with arguments. Note that the macro name and definition are placed between double quotes because otherwise the spaces would indicate a new option.

```
ccmcs --define="MAX(A,B)=((A) > (B) ? (A) : (B))" test.asm
```

## Related information

Control program option **--option-file** (Specify an option file)

# Control program option: --dep-file

## Menu entry

-

## Command line syntax

**`--dep-file`**`[`**`=`**`*file*]`

## Description

With this option you tell the assembler to generate dependency lines that can be used in a Makefile. The dependency information will be generated in addition to the normal output file.

By default, the information is written to a file with extension `.d` (one for every input file). When you specify a filename, all dependencies will be combined in the specified file.

## Example

```
ccmcs --dep-file=test.dep -t test.asm
```

The assembler assembles the file `test.asm`, which results in the output file `test.o`, and generates dependency lines in the file `test.dep`.

## Related information

-

# Control program option: --diag

## Menu entry

1. From the **Window** menu, select **Show View » Other » TASKING » Problems**.

   *The Problems view is added to the current perspective.*

2. In the Problems view right-click on a message.

   *A popup menu appears.*

3. Select **Detailed Diagnostics Info**.

   *A dialog box appears with additional information.*

## Command line syntax

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

You can set the following output formats:

| | |
|---|---|
| **html** | HTML output. |
| **rtf** | Rich Text Format. |
| **text** | ASCII text. |

Default format: text

## Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to stdout (normally your screen) and in the format you specify. You can specify the following formats: **html**, **rtf** or **text** (default). To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas.

## Example

To display an explanation of message number 103, enter:

```
ccmcs --diag=103
```

This results in message 103 with explanation.

To write an explanation of all errors and warnings in HTML format to file `ccerrors.html`, use redirection and enter:

```
ccmcs --diag=html:all > ccerrors.html
```

### Related information

-

## Control program option: --dry-run (-n)

### Menu entry

-

### Command line syntax

**`--dry-run`**

**`-n`**

### Description

With this option you put the control program in verbose mode. The control program prints the invocations of the tools it would use to process the files without actually performing the steps.

### Related information

Control program option **--verbose** (Verbose output)

## Control program option: --dwarf-version

### Menu entry

-

### Command line syntax

**`--dwarf-version=`**`{`**`2`**`|`**`3`**`}`

Default: **`3`**

### Description

With this option you tell the assembler which DWARF debug version to generate, DWARF2 or DWARF3 (default).

### Related information

-

# Control program option: --error-file

## Menu entry

-

## Command line syntax

**`--error-file`**

## Description

With this option the control program tells the assembler and linker to redirect error messages to a file.

## Example

To write errors to error files instead of stderr, enter:

```
ccmcs --error-file test.asm
```

## Related information

Control Program option **--warnings-as-errors** (Treat warnings as errors)

# Control program option: --help (-?)

## Menu entry

-

## Command line syntax

**`--help`**`[`**`=`**`*item*]`

**`-?`**

You can specify the following argument:

| | |
|---|---|
| **options** | Show extended option descriptions |

## Description

Displays an overview of all command line options. When you specify the argument **options** you can list detailed option descriptions.

## Example

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

```
ccmcs -?
ccmcs --help
ccmcs
```

To see a detailed description of the available options, enter:

```
ccmcs --help=options
```

## Related information

-

# Control program option: --include-directory (-I)

## Menu entry

1. Select **Assembler » Include Paths**.

   *The Include paths box shows the directories that are added to the search path for include files.*

2. To define a new directory for the search path, click on the **Add** button in the **Include paths** box.

3. Type or select a path.

Use the **Edit** and **Delete** button to change a path or to remove a path from the list.

## Command line syntax

```
--include-directory=path,...
```

```
-Ipath,...
```

## Description

With this option you can specify the path where your include files are located. A relative path will be relative to the current directory.

The control program passes this option to the assembler.

## Example

Suppose that the assembly source file `test.asm` contains the following lines:

```
.INCLUDE 'myinc.inc'
```

You can call the control program as follows:

```
ccmcs --include-directory=c:\proj\include test.asm
```

First the assembler looks for the file `myinc.inc` in the directory where `test.asm` is located. If it does not find the file, it looks in the directory `c:\proj\include` (this option). If the file is still not found, the assembler searches in the environment variable and then in the default include directory.

## Related information

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

Assembler option **--include-file** (Include file at the start of the input file)

# Control program option: --keep-output-files (-k)

## Menu entry

Eclipse *always* removes generated output files when an error occurs.

## Command line syntax

**`--keep-output-files`**

**`-k`**

## Description

If an error occurs during the assembling or linking process, the resulting output file may be incomplete or incorrect. With this option you keep the generated output files when an error occurs.

By default the control program removes generated output files when an error occurs. This is useful when you use the make utility. If the erroneous files are not removed, the make utility may process corrupt files on a subsequent invocation.

Use this option when you still want to use the generated files. For example when you know that a particular error does not result in a corrupt file, or when you want to inspect the output file, or send it to Altium support.

The control program passes this option to the assembler and linker.

## Example

```
ccmcs --keep-output-files test.asm
```

When an error occurs during assembling or linking, the erroneous generated output files will not be removed.

## Related information

Assembler option **--keep-output-files**

Linker option **--keep-output-files**

# Control program option: --keep-temporary-files (-t)

## Menu entry

1. Select **Global Options**.
2. Enable the option **Keep temporary files**.

## Command line syntax

**`--keep-temporary-files`**

**`-t`**

## Description

By default, the control program removes intermediate files like the `.o` file (result of the assembler phase).

With this option you tell the control program to keep temporary files it generates during the creation of the absolute object file.

## Example

```
ccmcs --keep-temporary-files test.asm
```

The control program keeps all intermediate files it generates while creating the absolute object file `test.elf`.

## Related information

-

# Control program option: --list-files

## Menu entry

-

## Command line syntax

**`--list-files`**`[=`*`file`*`]`

Default: no list files are generated

## Description

With this option you tell the assembler via the control program to generate a list file for each specified input file. A list file shows the generated object code and the relative addresses. Note that the assembler generates a relocatable object file with relative addresses.

With the optional *file* you can specify a name for the list file. This is only possible if you specify only one input file to the control program. If you do not specify a file name, or you specify more than one input file, the control program names the generated list file(s) after the specified input file(s) with extension `.lst`.

Note that object files and library files are not counted as input files.

## Related information

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

Assembler option **--list-format** (Format list file)

# Control program option: --lsl-core

## Menu entry

1. Expand **C/C++ Build** and select **Processor**.

2. From the **Select core** list, select a processor core.

## Command line syntax

**--lsl-core=***MCS-core*

You can specify the following MCS cores:

| | |
|---|---|
| **mcs00** | MCS core 0 |
| **mcs01** | MCS core 1 |
| **mcs02** | MCS core 2 |
| **mcs03** | MCS core 3 |

Default: **mcs00**

## Description

With this option you can specify the LSL core architecture the code is intended for. For example, the file `tc27x.lsl` in the `include.lsl` directory, contains a description of derivative `tc27x` and the supported MCS cores.

## Example

To link objects for the MCS core `mcs01`, enter:

```
ccmcs --cpu=tc27x --lsl-core=mcs01 test.asm
```

This results in the following invocation of the tools:

```
+ asmcs -Ctc27x -o test.o test.asm
+ lcms  -o test.elf -dtc27x.lsl --non-romable
        --user-provided-initialization-code -D__LINKONLY__
        -DCSA=0 --core=mpe:mcs01 --map-file test.o
```

## Related information

Linker option **--core** (Specify LSL core)

# Control program option: --lsl-file (-d)

## Menu entry

An LSL file can be generated when you create your TriCore project in Eclipse:

1. From the **File** menu, select **File » New » TASKING TriCore C/C++ Project**.

   *The New C/C++ Project wizard appears.*

2. Fill in the project settings in each dialog and click **Next >** until the **TriCore Project Settings** appear.

3. Enable the option **Add linker script file to the project** and click **Finish**.

   *Eclipse creates your project and the file project`.lsl` in the project directory.*

The LSL file can be specified in the Properties dialog:

1. Select **Linker » Script File**.

2. Specify a LSL file in the **Linker script file (.lsl)** field.

## Command line syntax

```
--lsl-file=file,...
```

```
-dfile,...
```

## Description

A linker script file contains vital information about the core for the locating phase of the linker. A linker script file is coded in LSL and contains the following types of information:

- the architecture definition describes the core's hardware architecture.
- the memory definition describes the physical memory available in the system.
- the section layout definition describes how to locate sections in memory.

With this option you specify a linker script file via the control program to the linker. If you do not specify this option, the linker uses a default script file. You can specify the existing file *target*`.lsl` or the name of a manually written linker script file. You can use this option multiple times. The linker processes the LSL files in the order in which they appear on the command line.

## Related information

Section 3.6, *Controlling the Linker with a Script*

## Control program option: --make-target

### Menu entry

-

### Command line syntax

**`--make-target=`***`name`*

### Description

With this option you can overrule the default target name in the make dependencies generated by the option **--dep-file**. The default target name is the basename of the input file, with extension `.o`.

### Example

```
ccmcs --preprocess=+make --make-target=../mytarget.o test.asm
```

The assembler generates dependency lines with the default target name `../mytarget.o` instead of `test.o`.

### Related information

Control program option **--dep-file** (Generate dependencies in a file)

# Control program option: --no-map-file

## Menu entry

1. Select **Linker » Map File**.

2. Disable the option **Generate map file**.

## Command line syntax

```
--no-map-file
```

## Description

By default the control program tells the linker to generate a linker map file.

A linker map file is a text file that shows how the linker has mapped the sections and symbols from the various object files (`.o`) to the linked object file. A locate part shows the absolute position of each section. External symbols are listed per space with their absolute address, both sorted on symbol and sorted on address.

With this option you prevent the generation of a map file.

## Related information

-

# Control program option: --no-warnings (-w)

## Menu entry

1. Select **Assembler » Diagnostics**.

   *The Suppress warnings box shows the warnings that are currently suppressed.*

2. To suppress a warning, click on the **Add** button in the **Suppress warnings** box.

3. Enter the numbers, separated by commas or as a range, of the warnings you want to suppress (for example `537,538`). Or you can use the **Add** button multiple times.

4. To suppress all warnings, enable the option **Suppress all warnings**.

> Use the **Edit** and **Delete** button to change a warning number or to remove a number from the list.

## Command line syntax

```
--no-warnings[=number[-number],...]
```

```
-w[number[-number],...]
```

## Description

With this option you can suppresses all warning messages for the various tools or specific control program warning messages.

On the command line this option works as follows:

- If you do not specify this option, all warnings are reported.
- If you specify this option but without numbers, all warnings of all tools are suppressed.
- If you specify this option with a number or a range, only the specified control program warnings are suppressed. You can specify the option **--no-warnings=***number* multiple times.

## Example

To suppress all warnings for all tools, enter:

```
ccmcs test.asm --no-warnings
```

## Related information

Control program option **--warnings-as-errors** (Treat warnings as errors)

# Control program option: --option-file (-f)

## Menu entry

-

## Command line syntax

**--option-file=***file*,...

**-f** *file*,...

## Description

This option is primarily intended for command line use. Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the control program.

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

You can specify the option **--option-file** multiple times.

### Format of an option file

- Multiple arguments on one line in the option file are allowed.
- To include whitespace in an argument, surround the argument with single or double quotes.
- If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

  ```
  "This has a single quote ' embedded"

  'This has a double quote " embedded'

  'This has a double quote " and a single quote '"' embedded"
  ```

- When a text line reaches its length limit, use a **** to continue the line. Whitespace between quotes is preserved.

  ```
  "This is a continuation \
  line"

           -> "This is a continuation line"
  ```

- It is possible to nest command line files up to 25 levels.

## Example

Suppose the file `myoptions` contains the following lines:

```
--debug-info
--define=DEMO=1
test.asm
```

Specify the option file to the control program:

```
ccmcs --option-file=myoptions
```

This is equivalent to the following command line:

```
ccmcs --debug-info --define=DEMO=1 test.asm
```

## Related information

-

# Control program option: --output (-o)

## Menu entry

Eclipse always uses the project name as the basename for the output file.

## Command line syntax

**--output=***file*

**-o** *file*

## Description

By default, the control program generates a file with the same basename as the first specified input file. With this option you specify another name for the resulting absolute object file.

## Example

```
ccmcs test.asm prog.asm
```

The control program generates an ELF/DWARF object file (default) with the name `test.elf`.

To generate the file `result.elf`:

```
ccmcs --output=result.elf test.asm prog.asm
```

## Related information

Linker option **--output** (Output file)

Linker option **--chip-output** (Generate an output file for each chip)

# Control program option: --pass (-W)

## Menu entry

1. Select **Assembler » Miscellaneous** or **Linker » Miscellaneous**.

2. Add an option to the **Additional options** field.

   *Be aware that the options in the option file are added to the options you have set in the other pages. Only in extraordinary cases you may want to use them in combination. The assembler options are preceded by* ***-Wa*** *and the linker options are preceded by* ***-Wl****.*

## Command line syntax

| | | |
|---|---|---|
| **--pass-assembler=***option* | **-Wa***option* | Pass option directly to the assembler |
| **--pass-linker=***option* | **-Wl***option* | Pass option directly to the linker |

## Description

With this option you tell the control program to call a tool with the specified option. The control program does not use or interpret the option itself, but specifies it directly to the tool which it calls.

## Example

To pass the option **--verbose** directly to the linker, enter:

```
ccmcs --pass-linker=--verbose test.asm
```

## Related information

-

# Control program option: --processors

## Menu entry

1. From the **Window** menu, select **Preferences**.

   *The Preferences dialog appears.*

2. Select **TASKING » MCS**.

3. Click the **Add** button to add additional processor definition files.

## Command line syntax

**--processors=***file*

## Description

With this option you can specify an additional XML file with processor definitions.

The standard list of supported processors is defined in the file `processors.xml`. This file defines for each processor its full name (for example, TC27X), the base CPU name (for example, tc27x) and the core settings (for example, mcs).

The control program reads the specified *file* after the file `processors.xml` in the product's `etc` directory. Additional XML files can override processor definitions made in XML files that are read before.

Multiple **--processors** options are allowed.

Eclipse generates a **--processors** option in the makefiles for each specified XML file.

## Example

Specify an additional processor definition file (suppose `processors-new.xml` contains a new processor `MCSNEW`):

```
ccmcs --processors=processors-new.xml --cpu=MCSNEW test.c
```

## Related information

Control program option **--cpu** (Select processor)

# Control program option: --verbose (-v)

## Menu entry

1. Select **Global Options**.
2. Enable the option **Verbose mode of control program**.

## Command line syntax

```
--verbose
```

```
-v
```

## Description

With this option you put the control program in verbose mode. The control program performs it tasks while it prints the steps it performs to stdout.

## Related information

Control program option **--dry-run** (Verbose output and suppress execution)

## Control program option: --version (-V)

### Menu entry

-

### Command line syntax

```
--version
```

```
-V
```

### Description

Display version information. The control program ignores all other options or input files.

### Related information

-

# Control program option: --warnings-as-errors

## Menu entry

1. Select **Global Options**.

2. Enable the option **Treat warnings as errors**.

## Command line syntax

```
--warnings-as-errors[=number[-number],...]
```

## Description

If one of the tools encounters an error, it stops processing the file(s). With this option you tell the tools to treat warnings as errors or treat specific control program warning messages as errors:

- If you specify this option but without numbers, all warnings are treated as errors.

- If you specify this option with a number or a range, only the specified control program warnings are treated as an error. You can specify the option **--warnings-as-errors=***number* multiple times.

Use one of the **--pass-***tool* options to pass this option directly to a tool when a specific warning for that tool must be treated as an error. For example, use **--pass-assembler=--warnings-as-errors=***number* to treat a specific assembler warning as an error.

## Related information

Control program option **--no-warnings** (Suppress some or all warnings)

Control program option **--pass** (Pass option to tool)

## 5.4. Make Utility Options

When you build a project in Eclipse, Eclipse generates a makefile and uses the make utility **mkmcs** to build all your files. However, you can also use the make utility directly from the command line to build your project.

The invocation syntax is:

```
mkmcs [option...] [target...] [macro=def]
```

This section describes all options for the make utility. The make utility is a command line tool so there are no equivalent options in Eclipse.

For detailed information about the make utility and using makefiles see Section 4.2, *Make Utility mkmcs*.

# Defining Macros

## Command line syntax

*macro_name*[**=***macro_definition*]

## Description

With this argument you can define a macro and specify it to the make utility.

A macro definition remains in existence during the execution of the makefile, even when the makefile recursively calls the make utility again. In the recursive call, the macro acts as an environment variable. This means that it is overruled by definitions in the recursive call. Use the option **-e** to prevent this.

You can specify as many macros as you like. If the command line exceeds the limit of the operating system, you can define the macros in an option file which you then must specify to the make utility with the option **-m**) *file*.

Defining macros on the command line is, for example, useful in combination with conditional processing as shown in the example below.

## Example

Consider the following makefile with conditional rules to build a demo program and a real program:

```
ifdef DEMO       # the value of DEMO is of no importance
   real.elf : demo.o main.o
              lmcs demo.o main.o -lc -lfp
else
   real.elf : real.o main.o
              lmcs real.o main.o -lc -lfp
endif
```

You can now use a macro definition to set the DEMO flag:

```
mkmcs real.elf DEMO=1
```

In both cases the absolute object file `real.elf` is created but depending on the DEMO flag it is linked with `demo.o` or with `real.o`.

## Related information

Make utility option **-e** (Environment variables override macro definitions)

Make utility option **-m** (Name of invocation file)

# Make utility option: -?

## Command line syntax

**`-?`**

## Description

Displays an overview of all command line options.

## Example

The following invocation displays a list of the available command line options:

```
mkmcs -?
```

## Related information

-

## Make utility option: -a

### Command line syntax

```
-a
```

### Description

Normally the make utility rebuilds only those files that are out of date. With this option you tell the make utility to rebuild *all* files, without checking whether they are out of date.

### Example

```
mkmcs -a
```

Rebuilds all your files, regardless of whether they are out of date or not.

### Related information

-

# Make utility option: -c

## Command line syntax

```
-c
```

## Description

Eclipse uses this option when you create sub-projects. In this case the make utility calls another instance of the make utility for the sub-project. With the option **-c**, the make utility runs as a child process of the current make.

The option **-c** overrules the option **-err**.

## Example

```
mkmcs -c
```

The make utility runs its commands as a child processes.

## Related information

Make utility option **-err** (Redirect error message to file)

# Make utility option: -D / -DD

## Command line syntax

```
-D
-DD
```

## Description

With the option **-D** the make utility prints every line of the makefile to standard output as it is read by **mkmcs**.

With the option **-DD** not only the lines of the makefile are printed but also the lines of the `mkmcs.mk` file (implicit rules).

## Example

```
mkmcs -D
```

Each line of the makefile that is read by the make utility is printed to standard output (usually your screen).

## Related information

-

# Make utility option: -d/ -dd

## Command line syntax

```
-d
-dd
```

## Description

With the option **-d** the make utility shows which files are out of date and thus need to be rebuild. The option **-dd** gives more detail than the option **-d**.

## Example

```
mkmcs -d
```

Shows which files are out of date and rebuilds them.

## Related information

-

# Make utility option: -e

## Command line syntax

```
-e
```

## Description

If you use macro definitions, they may overrule the settings of the environment variables. With the option **-e**, the settings of the environment variables are used even if macros define otherwise.

## Example

```
mkmcs -e
```

The make utility uses the settings of the environment variables regardless of macro definitions.

## Related information

-

# Make utility option: -err

## Command line syntax

**-err** *file*

## Description

With this option the make utility redirects error messages and verbose messages to a specified file.

With the option **-s** the make utility only displays error messages.

## Example

```
mkmcs -err error.txt
```

The make utility writes messages to the file `error.txt`.

## Related information

Make utility option **-s** (Do not print commands before execution)

Make utility option **-c** (Run as child process)

# Make utility option: -f

## Command line syntax

```
-f my_makefile
```

## Description

By default the make utility uses the file `makefile` to build your files.

With this option you tell the make utility to use the specified file instead of the file `makefile`. Multiple **-f** options act as if all the makefiles were concatenated in a left-to-right order.

If you use '-' instead of a makefile name it means that the information is read from `stdin`.

## Example

```
mkmcs -f mymake
```

The make utility uses the file `mymake` to build your files.

## Related information

-

## Make utility option: -G

### Command line syntax

**-G** *path*

### Description

Normally you must call the make utility from the directory where your makefile and other files are stored.

With the option **-G** you can call the make utility from within another directory. The *path* is the path to the directory where your makefile and other files are stored and can be absolute or relative to your current directory.

### Example

Suppose your makefile and other files are stored in the directory `..\myfiles`. You can call the make utility, for example, as follows:

```
mkmcs -G ..\myfiles
```

### Related information

-

# Make utility option: -i

## Command line syntax

```
-i
```

## Description

When an error occurs during the make process, the make utility exits with a certain exit code.

With the option **-i**, the make utility exits without an error code, even when errors occurred.

## Example

```
mkmcs -i
```

The make utility exits without an error code, even when an error occurs.

## Related information

-

# Make utility option: -K

## Command line syntax

```
-K
```

## Description

With this option the make utility keeps temporary files it creates during the make process. The make utility stores temporary files in the directory that you have specified with the environment variable TMPDIR or in the default 'temp' directory of your system when the TMPDIR environment variable is not specified.

## Example

```
mkmcs -K
```

The make utility preserves all temporary files.

## Related information

-

# Make utility option: -k

## Command line syntax

**-k**

## Description

When during the make process the make utility encounters an error, it stops rebuilding your files.

With the option **-k**, the make utility only stops building the target that produced the error. All other targets defined in the makefile are built.

## Example

```
mkmcs -k
```

If the make utility encounters an error, it stops building the current target but proceeds with the other targets that are defined in the makefile.

## Related information

Make utility option **-S** (Undo the effect of **-k**)

# Make utility option: -m

## Command line syntax

**-m** *file*

## Description

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

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

You can specify the option **-m** multiple times.

If you use '-' instead of a filename it means that the options are read from `stdin`.

### Format of an option file

- Multiple arguments on one line in the option file are allowed.
- To include whitespace in an argument, surround the argument with single or double quotes.
- If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

  ```
  "This has a single quote ' embedded"
  ```

  ```
  'This has a double quote " embedded'
  ```

  ```
  'This has a double quote " and a single quote '"' embedded"
  ```

  Note that adjacent strings are concatenated.

- When a text line reaches its length limit, use a **\** to continue the line. Whitespace between quotes is preserved.

  ```
  "This is a continuation \
  line"
  ```

  ```
           -> "This is a continuation line"
  ```

- It is possible to nest command line files up to 25 levels.

## Example

Suppose the file `myoptions` contains the following lines:

```
-k
-err errors.txt
test.elf
```

Specify the option file to the make utility:

```
mkmcs -m myoptions
```

This is equivalent to the following command line:

```
mkmcs -k -err errors.txt test.elf
```

## Related information

-

# Make utility option: -n

## Command line syntax

**`-n`**

## Description

With this option you tell the make utility to perform a dry run. The make utility shows what it would do but does not actually perform these tasks.

This option is for example useful to quickly inspect what would happen if you call the make utility.

## Example

```
mkmcs -n
```

The make utility does not perform any tasks but displays what it would do if called without the option **-n**.

## Related information

Make utility option **-s** (Do not print commands before execution)

# Make utility option: -p

## Command line syntax

**`-p`**

## Description

Normally, if a command in a target rule in a makefile returns an error or when the target construction is interrupted, the make utility removes that target file. With this option you tell the make utility to make all target files precious. This means that all dependency files are never removed.

## Example

```
mkmcs -p
```

The make utility never removes target dependency files.

## Related information

Special target `.PRECIOUS` in Section 4.2.2.1, *Targets and Dependencies*

# Make utility option: -q

## Command line syntax

**`-q`**

## Description

With this option the make utility does not perform any tasks but only returns an exit code. A zero status indicates that all target files are up to date, a non-zero status indicates that some or all target files are out of date.

## Example

```
mkmcs -q
```

The make utility only returns an error code that indicates whether all target files are up to date or not. It does not rebuild any files.

## Related information

-

# Make utility option: -r

## Command line syntax

```
-r
```

## Description

When you call the make utility, it first reads the implicit rules from the file `mkmcs.mk`, then it reads the makefile with the rules to build your files. (The file `mkmcs.mk` is located in the `\etc` directory of the toolset.)

With this option you tell the make utility not to read `mkmcs.mk` and to rely fully on the make rules in the makefile.

## Example

```
mkmcs -r
```

The make utility does not read the implicit make rules in `mkmcs.mk`.

## Related information

-

# Make utility option: -S

## Command line syntax

```
-S
```

## Description

With this option you cancel the effect of the option **-k**. This is only necessary in a recursive make where the option **-k** might be inherited from the top-level make via MAKEFLAGS or if you set the option **-k** in the environment variable MAKEFLAGS.

With this option you tell the make utility not to read `mkmcs.mk` and to rely fully on the make rules in the makefile.

## Example

```
mkmcs -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 **mkmcs** in the makefile.

## Related information

Make utility option **-k** (On error, abandon the work for the current target only)

# Make utility option: -s

## Command line syntax

**`-s`**

## Description

With this option you tell the make utility to perform its tasks without printing the commands it executes. Error messages are normally printed.

## Example

```
mkmcs -s
```

The make utility rebuilds your files but does not print the commands it executes during the make process.

## Related information

Make utility option **-n** (Perform a dry run)

# Make utility option: -t

## Command line syntax

**`-t`**

## Description

With this option you tell the make utility to *touch* the target files, bringing them up to date, rather than performing the rules to rebuild them.

## Example

```
mkmcs -t
```

The make utility updates out-of-date files by giving them a new date and time stamp. The files are not actually rebuild.

## Related information

-

# Make utility option: -time

## Command line syntax

**`-time`**

## Description

With this option you tell the make utility to display the current date and time on standard output.

## Example

```
mkmcs -time
```

The make utility displays the current date and time and updates out-of-date files.

## Related information

-

# Make utility option: -V

## Command line syntax

```
-V
```

## Description

Display version information. The make utility ignores all other options or input files.

## Related information

-

# Make utility option: -W

## Command line syntax

**`-W`** *`target`*

## Description

With this option the make utility considers the specified target file always as up to date and will not rebuild it.

## Example

```
mkmcs -W test.elf
```

The make utility rebuilds out of date targets in the makefile except the file `test.elf` which is considered now as up to date.

## Related information

-

# Make utility option: -w

## Command line syntax

```
-w
```

## Description

With this option the make utility sends error messages and verbose messages to standard output. Without this option, the make utility sends these messages to standard error.

> This option is only useful on UNIX systems.

## Example

```
mkmcs -w
```

The make utility sends messages to standard out instead of standard error.

## Related information

-

# Make utility option: -x

## Command line syntax

**`-x`**

## Description

With this option the make utility shows extended error messages. Extended error messages give more detailed information about the exit status of the make utility after errors.

## Example

```
mkmcs -x
```

If errors occur, the make utility gives extended information.

## Related information

-

## 5.5. Parallel Make Utility Options

When you build a project in Eclipse, Eclipse generates a makefile and uses the make utility **amk** to build all your files. However, you can also use the make utility directly from the command line to build your project.

The invocation syntax is:

```
amk [option...] [target...] [macro=def]
```

This section describes all options for the parallel make utility.

For detailed information about the parallel make utility and using makefiles see Section 4.3, *Make Utility amk*.

# Parallel make utility option: --always-rebuild (-a)

## Command line syntax

**`--always-rebuild`**

**`-a`**

## Description

Normally the make utility rebuilds only those files that are out of date. With this option you tell the make utility to rebuild *all* files, without checking whether they are out of date.

## Example

```
amk -a
```

Rebuilds all your files, regardless of whether they are out of date or not.

## Related information

-

# Parallel make utility option: --change-dir (-G)

## Command line syntax

**`--change-dir=`**`path`

**`-G`** `path`

## Description

Normally you must call the make utility from the directory where your makefile and other files are stored.

With the option **-G** you can call the make utility from within another directory. The *path* is the path to the directory where your makefile and other files are stored and can be absolute or relative to your current directory.

The macro `SUBDIR` is defined with the value of *path*.

## Example

Suppose your makefile and other files are stored in the directory `..\myfiles`. You can call the make utility, for example, as follows:

```
amk -G ..\myfiles
```

## Related information

-

# Parallel make utility option: --diag

## Command line syntax

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

You can set the following output formats:

| | |
|---|---|
| **html** | HTML output. |
| **rtf** | Rich Text Format. |
| **text** | ASCII text. |

Default format: text

## Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to stdout (normally your screen) and in the format you specify. You can specify the following formats: **html**, **rtf** or **text** (default). To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas.

## Example

To display an explanation of message number 169, enter:

```
amk --diag=169
```

This results in the following message and explanation:

```
F169: target '%s' returned exit code %d

An error occured while executing one of the commands
of the target, and -k option is not specified.
```

To write an explanation of all errors and warnings in HTML format to file `amkerrors.html`, use redirection and enter:

```
amk --diag=html:all > amkerrors.html
```

## Related information

-

# Parallel make utility option: --dry-run (-n)

## Command line syntax

**`--dry-run`**

**`-n`**

## Description

With this option you tell the make utility to perform a dry run. The make utility shows what it would do but does not actually perform these tasks.

This option is for example useful to quickly inspect what would happen if you call the make utility.

## Example

```
amk -n
```

The make utility does not perform any tasks but displays what it would do if called without the option **-n**.

## Related information

Parallel make utility option **-s** (Do not print commands before execution)

# Parallel make utility option: --help (-? / -h)

## Command line syntax

**--help**[**=***item*]

**-h**

**-?**

You can specify the following arguments:

| | |
|---|---|
| **options** | Show extended option descriptions |

## Description

Displays an overview of all command line options. When you specify the argument **options** you can list detailed option descriptions.

## Example

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

```
amk -?
amk --help
```

To see a detailed description of the available options, enter:

```
amk --help=options
```

## Related information

-

# Parallel make utility option: --jobs (-j) / --jobs-limit (-J)

## Menu

1. From the **Project** menu, select **Properties for**

   *The Properties dialog appears.*

2. In the left pane, select **C/C++ Build**.

   *In the right pane the C/C++ Build page appears.*

3. On the Behaviour tab, select **Use parallel build**.

4. You can specify the number of parallel jobs, or you can use an optimal number of jobs. In the last case, **amk** will fork as many jobs in parallel as cores are available.

## Command line syntax

```
--jobs[=number]
-j[number]
```

```
--jobs-limit[=number]
-J[number]
```

## Description

When these options you can limit the number of parallel jobs. The default is 1. Zero means no limit. When you omit the *number*, **amk** uses the number of cores detected.

Option **-J** is the same as **-j**, except that the number of parallel jobs is limited by the number of cores detected.

## Example

```
amk -j3
```

Limit the number of parallel jobs to 3.

## Related information

-

# Parallel make utility option: --keep-going (-k)

## Command line syntax

**`--keep-going`**

**`-k`**

## Description

When during the make process the make utility encounters an error, it stops rebuilding your files.

With the option **-k**, the make utility only stops building the target that produced the error. All other targets defined in the makefile are built.

## Example

```
amk -k
```

If the make utility encounters an error, it stops building the current target but proceeds with the other targets that are defined in the makefile.

## Related information

-

# Parallel make utility option: --list-targets (-l)

## Command line syntax

**`--list-targets`**

**`-l`**

## Description

With this option, the make utility lists all "primary" targets that are out of date.

## Example

```
amk -l
list of targets
```

## Related information

-

# Parallel make utility option: --makefile (-f)

## Command line syntax

**`--makefile=`***`my_makefile`*

**`-f`** *`my_makefile`*

## Description

By default the make utility uses the file `makefile` to build your files.

With this option you tell the make utility to use the specified file instead of the file `makefile`. Multiple **-f** options act as if all the makefiles were concatenated in a left-to-right order.

If you use '-' instead of a makefile name it means that the information is read from `stdin`.

## Example

```
amk -f mymake
```

The make utility uses the file `mymake` to build your files.

## Related information

-

# Parallel make utility option: --no-warnings (-w)

## Command line syntax

```
--no-warnings[=number,...]
```

```
-w[number,...]
```

## Description

With this option you can suppresses all warning messages or specific warning messages.

On the command line this option works as follows:

- If you do not specify this option, all warnings are reported.
- If you specify this option but without numbers, all warnings are suppressed.
- If you specify this option with a number, only the specified warning is suppressed. You can specify the option **--no-warnings=***number* multiple times.

## Example

To suppress warnings 751 and 756, enter:

```
amk --no-warnings=751,756
```

## Related information

Parallel make utility option **--warnings-as-errors** (Treat warnings as errors)

# Parallel make utility option: --silent (-s)

## Command line syntax

**`--silent`**

**`-s`**

## Description

With this option you tell the make utility to perform its tasks without printing the commands it executes. Error messages are normally printed.

## Example

```
amk -s
```

The make utility rebuilds your files but does not print the commands it executes during the make process.

## Related information

Parallel make utility option **-n** (Perform a dry run)

# Parallel make utility option: --version (-V)

## Command line syntax

**`--version`**

**`-V`**

## Description

Display version information. The make utility ignores all other options or input files.

## Related information

-

# Parallel make utility option: --warnings-as-errors

## Command line syntax

```
--warnings-as-errors[=number,...]
```

## Description

If the make utility encounters an error, it stops. When you use this option without arguments, you tell the make utility to treat all warnings as errors. This means that the exit status of the make utility will be non-zero after one or more warnings. As a consequence, the make utility now also stops after encountering a warning.

You can also limit this option to specific warnings by specifying a comma-separated list of warning numbers.

## Related information

Parallel make utility option **--no-warnings** (Suppress some or all warnings)

# 5.6. Archiver Options

The archiver and library maintainer **armcs** 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:

```
armcs key_option [sub_option...] library [object_file]
```

This section describes all options for the archiver. Some suboptions can only be used in combination with certain key options. They are described together. Suboptions that can always be used are described separately.

For detailed information about the archiver, see Section 4.4, *Archiver*.

## Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (**-**) character, long option names always begin with two minus (**--**) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

## Overview of the options of the archiver utility

The following archiver options are available:

| Description | Option | Sub-option |
|---|---|---|
| **Main functions (key options)** | | |
| Replace or add an object module | **-r** | **-a -b -c -u -v** |
| Extract an object module from the library | **-x** | **-o -v** |
| Delete object module from library | **-d** | **-v** |
| Move object module to another position | **-m** | **-a -b -v** |
| Print a table of contents of the library | **-t** | **-s0 -s1** |
| Print object module to standard output | **-p** | |
| **Sub-options** | | |
| Append or move new modules after existing module *name* | **-a** *name* | |
| Append or move new modules before existing module *name* | **-b** *name* | |
| Create library without notification if library does not exist | **-c** | |
| Preserve last-modified date from the library | **-o** | |
| Print symbols in library modules | **-s**{**0**\|**1**} | |
| Replace only newer modules | **-u** | |
| Verbose | **-v** | |
| **Miscellaneous** | | |

| **Description** | **Option** | **Sub-option** |
|---|---|---|
| Display options | **-?** | |
| Display version header | **-V** | |
| Read options from *file* | **-f** *file* | |
| Suppress warnings above level *n* | **-w***n* | |

# Archiver option: --delete (-d)

## Command line syntax

```
--delete [--verbose]
```

```
-d [-v]
```

## Description

Delete the specified object modules from a library. With the suboption **--verbose** (**-v**) the archiver shows which files are removed.

| | | |
|---|---|---|
| **--verbose** | **-v** | Verbose: the archiver shows which files are removed. |

## Example

```
armcs --delete mylib.a obj1.o obj2.o
```

The archiver deletes `obj1.o` and `obj2.o` from the library `mylib.a`.

```
armcs -d -v mylib.a obj1.o obj2.o
```

The archiver deletes `obj1.o` and `obj2.o` from the library `mylib.a` and displays which files are removed.

## Related information

-

# Archiver option: --dump (-p)

## Command line syntax

**`--dump`**

**`-p`**

## Description

Print the specified object module(s) in the library to standard output.

This option is only useful when you redirect or pipe the output to other files or tools that serve your own purposes. Normally you do not need this option.

## Example

```
armcs --dump mylib.a obj1.o > file.o
```

The archiver prints the file `obj1.o` to standard output where it is redirected to the file `file.o`. The effect of this example is very similar to extracting a file from the library but in this case the 'extracted' file gets another name.

## Related information

-

# Archiver option: --extract (-x)

## Command line syntax

```
--extract [--modtime] [--verbose]
```

```
-x [-o] [-v]
```

## Description

Extract an existing module from the library.

| | | |
|---|---|---|
| **--modtime** | **-o** | Give the extracted object module the same date as the last-modified date that was recorded in the library. Without this suboption it receives the last-modified date of the moment it is extracted. |
| **--verbose** | **-v** | Verbose: the archiver shows which files are extracted. |

## Example

To extract the file `obj1.o` from the library `mylib.a`:

```
armcs --extract mylib.a obj1.o
```

If you do not specify an object module, all object modules are extracted:

```
armcs -x mylib.a
```

## Related information

-

# Archiver option: --help (-?)

## Command line syntax

**`--help`**`[=`*`item`*`]`

**`-?`**

You can specify the following argument:

| | |
|---|---|
| **options** | Show extended option descriptions |

## Description

Displays an overview of all command line options. When you specify the argument **options** you can list detailed option descriptions.

## Example

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

```
armcs -?
armcs --help
armcs
```

To see a detailed description of the available options, enter:

```
armcs --help=options
```

## Related information

-

# Archiver option: --move (-m)

## Command line syntax

**--move** [**-a** *posname*] [**-b** *posname*]

**-m** [**-a** *posname*] [**-b** *posname*]

## Description

Move the specified object modules to another position in the library.

The ordering of members in a library can make a difference in how programs are linked if a symbol is defined in more than one member.

By default, the specified members are moved to the end of the archive. Use the suboptions **-a** or **-b** to move them to a specified place instead.

| | | |
|---|---|---|
| **--after=***posname* | **-a** *posname* | Move the specified object module(s) after the existing module *posname*. |
| **--before=***posname* | **-b** *posname* | Move the specified object module(s) before the existing module *posname*. |

## Example

Suppose the library `mylib.a` contains the following objects (see option **--print**):

```
obj1.o
obj2.o
obj3.o
```

To move `obj1.o` to the end of `mylib.a`:

```
armcs --move mylib.a obj1.o
```

To move `obj3.o` just before `obj2.o`:

```
armcs -m -b obj3.o mylib.a obj2.o
```

The library `mylib.a` after these two invocations now looks like:

```
obj3.o
obj2.o
obj1.o
```

## Related information

Archiver option **--print (-t)** (Print library contents)

# Archiver option: --option-file (-f)

## Command line syntax

**--option-file=***file*

**-f** *file*

## Description

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

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

You can specify the option **--option-file** (**-f**) multiple times.

If you use '-' instead of a filename it means that the options are read from `stdin`.

### Format of an option file

- Multiple arguments on one line in the option file are allowed.
- To include whitespace in an argument, surround the argument with single or double quotes.
- If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

  ```
  "This has a single quote ' embedded"
  ```

  ```
  'This has a double quote " embedded'
  ```

  ```
  'This has a double quote " and a single quote '"' embedded"
  ```

- When a text line reaches its length limit, use a **** to continue the line. Whitespace between quotes is preserved.

  ```
  "This is a continuation \
  line"
  ```

  ```
           -> "This is a continuation line"
  ```

- It is possible to nest command line files up to 25 levels.

## Example

Suppose the file `myoptions` contains the following lines:

```
-x mylib.a obj1.o
-w5
```

Specify the option file to the archiver:

```
armcs --option-file=myoptions
```

This is equivalent to the following command line:

```
armcs -x mylib.a obj1.o -w5
```

## Related information

-

# Archiver option: --print (-t)

## Command line syntax

```
--print [--symbols=0|1]
```

```
-t [-s0|-s1]
```

## Description

Print a table of contents of the library to standard output. With the suboption **-s0** the archiver displays all symbols per object file.

| | | |
|---|---|---|
| **--symbols=0** | **-s0** | Displays per object the name of the object itself and all symbols in the object. |
| **--symbols=1** | **-s1** | Displays the symbols of all object files in the library in the form *library_name*:*object_name*:*symbol_name* |

## Example

```
armcs --print mylib.a
```

The archiver prints a list of all object modules in the library `mylib.a`:

```
armcs -t -s0 mylib.a
```

The archiver prints per object all symbols in the library.

## Related information

-

# Archiver option: --replace (-r)

## Command line syntax

```
--replace [--after=posname] [--before=posname][--create] [--newer-only] [--verbose]
```

```
-r [-a posname] [-b posname][-c] [-u] [-v]
```

## Description

You can use the option **--replace** (**-r**) for several purposes:

- Adding new objects to the library
- Replacing objects in the library with the same object of a newer date
- Creating a new library

The option **--replace** (**-r**) normally *adds* a new module to the library. However, if the library already contains a module with the specified name, the existing module is *replaced*. If you specify a library that does not exist, the archiver creates a new library with the specified name.

If you add a module to the library without specifying the suboption **-a** or **-b**, the specified module is added at the end of the archive. Use the suboptions **-a** or **-b** to insert them after/before a specified place instead.

| | | |
|---|---|---|
| **--after=***posname* | **-a** *posname* | Insert the specified object module(s) after the existing module *posname*. |
| **--before=***posname* | **-b** *posname* | Insert the specified object module(s) before the existing module *posname*. |
| **--create** | **-c** | Create a new library without checking whether it already exists. If the library already exists, it is overwritten. |
| **--newer-only** | **-u** | Insert the specified object module only if it is newer than the module in the library. |
| **--verbose** | **-v** | Verbose: the archiver shows which files are replaced. |

The suboptions **-a** or **-b** have no effect when an object is added to the library.

## Example

Suppose the library `mylib.a` contains the following object (see option **--print**):

```
obj1.o
```

To add `obj2.o` to the end of `mylib.a`:

```
armcs --replace mylib.a obj2.o
```

To insert `obj3.o` just before `obj2.o`:

```
armcs -r -b obj2.o mylib.a obj3.o
```

The library `mylib.a` after these two invocations now looks like:

```
obj1.o
obj3.o
obj2.o
```

### Creating a new library

To *create a new library file*, add an object file and specify a library that does not yet exist:

```
armcs --replace obj1.o newlib.a
```

The archiver creates the library `newlib.a` and adds the object `obj1.o` to it.

To *create a new library file and overwrite an existing library*, add an object file and specify an existing library with the supoption **-c**:

```
armcs -r -c obj1.o mylib.a
```

The archiver overwrites the library `mylib.a` and adds the object `obj1.o` to it. The new library `mylib.a` only contains `obj1.o`.

## Related information

Archiver option **--print** (**-t**) (Print library contents)

# Archiver option: --version (-V)

## Command line syntax

```
--version
```

```
-V
```

## Description

Display version information. The archiver ignores all other options or input files.

## Related information

-

# Archiver option: --warning (-w)

## Command line syntax

**`--warning=`***`level`*

**`-w`***`level`*

## Description

With this suboption you tell the archiver to suppress all warnings above the specified level. The level is a number between 0 - 9.

The level of a message is printed between parentheses after the warning number. If you do not use the **-w** option, the default warning level is 8.

## Example

To suppress warnings above level 5:

```
armcs --extract --warning=5 mylib.a obj1.o
```

## Related information

-

# Chapter 6. List File Formats

This chapter describes the format of the assembler list file and the linker map file.

## 6.1. Assembler List File Format

The assembler list file is an additional output file of the assembler that contains information about the generated code. For details on how to generate a list file, see Section 2.5, *Generating a List File*.

The list file consists of a page header and a source listing.

### Page header

The page header is repeated on every page:

```
TASKING VX-toolset for MCS: MCS assembler vx.yrz Build nnn SN 00000000
Title                                                            Page 1

ADDR CODE     CYCLES  LINE SOURCE LINE
```

The first line contains version information. The second line can contain a title which you can specify with the assembler control `$TITLE` and always contains a page number. The third line is empty and the fourth line contains the headings of the columns for the source listing.

With the assembler controls `$LIST ON/OFF`, `$PAGE`, and with the assembler option **--list-format** you can format the list file.

### Source listing

The following is a sample part of a listing. An explanation of the different columns follows below.

```
ADDR CODE     CYCLES  LINE SOURCE LINE
                         1        ; Module start
                         .
                         .
                        33        .sdecl  '.mcstext.proc.channel1',code
0000                    34        .sect   '.mcstext.proc.channel1'
                        35
                        36        .global process_channel1
0000                    37 process_channel1:       .type   func
0000 rrrr01A3  2    2   38        mrd     R3,chan1_par0
0004 rrrr01A2  2    4   39        mrd     R2,chan1_par1
                         .
                         .
0008                    44 buf:    .space  4
  |  RESERVED
0017
```

**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.

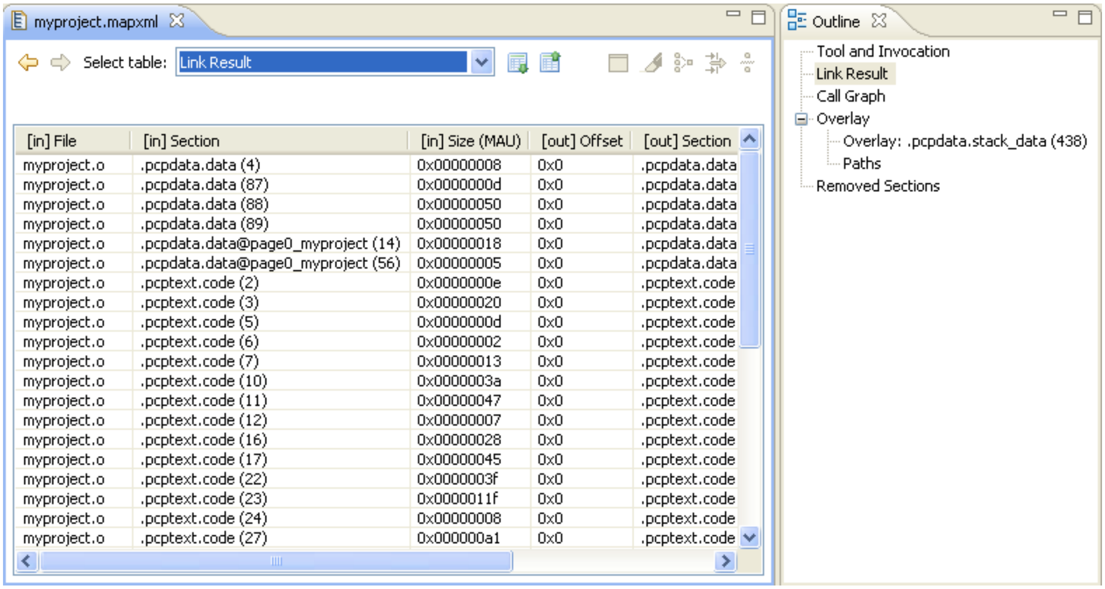
For the `.SET` and `.EQU` directives the `ADDR` and `CODE` columns do not apply. The symbol value is listed instead.

## 6.2. Linker Map File Format

The linker map file is an additional output file of the linker that shows how the linker has mapped the sections and symbols from the various object files (`.o`) to output sections. Locate information is not present, because that is not available for an MCS project. External symbols are listed per space with their absolute address, both sorted on symbol and sorted on address. For details on how to generate a map file, see Section 3.8, *Generating a Map File*.

With the linker option **--map-file-format** you can specify which parts of the map file you want to see.

In Eclipse the linker map file (*project*`.mapxml`) is generated in the output directory of the build configuration, usually `Debug` or `Release`. You can open the map file by double-clicking on the file name.



| [in] File | [in] Section | [in] Size (MAU) | [out] Offset | [out] Section |
|---|---|---|---|---|
| myproject.o | .pcpdata.data (4) | 0x00000008 | 0x0 | .pcpdata.data |
| myproject.o | .pcpdata.data (87) | 0x0000000d | 0x0 | .pcpdata.data |
| myproject.o | .pcpdata.data (88) | 0x00000050 | 0x0 | .pcpdata.data |
| myproject.o | .pcpdata.data (89) | 0x00000050 | 0x0 | .pcpdata.data |
| myproject.o | .pcpdata.data@page0_myproject (14) | 0x00000018 | 0x0 | .pcpdata.data |
| myproject.o | .pcpdata.data@page0_myproject (56) | 0x00000005 | 0x0 | .pcpdata.data |
| myproject.o | .pcptext.code (2) | 0x0000000e | 0x0 | .pcptext.code |
| myproject.o | .pcptext.code (3) | 0x00000020 | 0x0 | .pcptext.code |
| myproject.o | .pcptext.code (5) | 0x0000000d | 0x0 | .pcptext.code |
| myproject.o | .pcptext.code (6) | 0x00000002 | 0x0 | .pcptext.code |
| myproject.o | .pcptext.code (7) | 0x00000013 | 0x0 | .pcptext.code |
| myproject.o | .pcptext.code (10) | 0x0000003a | 0x0 | .pcptext.code |
| myproject.o | .pcptext.code (11) | 0x00000047 | 0x0 | .pcptext.code |
| myproject.o | .pcptext.code (12) | 0x00000007 | 0x0 | .pcptext.code |
| myproject.o | .pcptext.code (16) | 0x00000028 | 0x0 | .pcptext.code |
| myproject.o | .pcptext.code (17) | 0x00000045 | 0x0 | .pcptext.code |
| myproject.o | .pcptext.code (22) | 0x0000003f | 0x0 | .pcptext.code |
| myproject.o | .pcptext.code (23) | 0x0000011f | 0x0 | .pcptext.code |
| myproject.o | .pcptext.code (24) | 0x00000008 | 0x0 | .pcptext.code |
| myproject.o | .pcptext.code (27) | 0x000000a1 | 0x0 | .pcptext.code |

- Tool and Invocation
- Link Result
- Call Graph
- Overlay
  - Overlay: .pcpdata.stack_data (438)
  - Paths
- Removed Sections

Each page displays a part of the map file. You can use the drop-down list or the Outline view to navigate through the different tables and you can use the following buttons.

| Icon | Action | Description |
|---|---|---|
| ⇦ | Back | Goes back one page in the history list. |
| ⇨ | Forward | Goes forward one page in the history list. |
| ▦ | Next Table | Shows the next table from the drop-down list. |
| ▦ | Previous Table | Shows the previous table from the drop-down list. |

When you right-click in the view, a popup menu appears (for example, to reset the layout of a table). The meaning of the different parts is:

## Tool and Invocation

This part of the map file contains information about the linker, its version header information, binary location and which options are used to call it.

## Processed Files

This part of the map file shows all processed files. This also includes object files that are extracted from a library, with the symbol that led to the extraction.

## Link Result

This part of the map file shows per object file how the link phase has mapped the sections from the various object files (`.o`) to output sections.

| | |
|---|---|
| **[in] File** | The name of an input object file. |
| **[in] Section** | A section name and id from the input object file. The number between '( )' uniquely identifies the section. |
| **[in] Size** | The size of the input section. |
| **[out] Offset** | The offset relative to the start of the output section. |
| **[out] Section** | The resulting output section name and id. |
| **[out] Size** | The size of the output section. |

## Module Local Symbols

This part of the map file shows a table for each local scope within an object file. Each table has three columns, 1 the symbol name, 2 the address of the symbol and 3 the space where the symbol resides in. The table is sorted on symbol name within each space.

By default this part is not shown in the map file. You have to turn this part on manually with linker option **--map-file-format=+statics** (module local symbols).

## Cross References

This part of the map file lists all symbols defined in the object modules and for each symbol the object modules that contain a reference to the symbol are shown. Also, symbols that remain undefined are shown.

## Call Graph

This part is empty for the MCS.

## Overlay

This part is empty for the MCS.

## Processor and Memory

This part of the map file shows the processor and memory information of the linker script file.

By default this part is not shown in the map file. You have to turn this part on manually with linker option **--map-file-format=+lsl** (processor and memory info). You can print this information to a separate file with linker option **--lsl-dump**.

You can click the + or - sign to expand or collapse a part of the information.

## Removed Sections

This part of the map file shows the sections which are removed from the output file as a result of the optimization option to delete unreferenced sections and or duplicate code or constant data (linker option **--optimize=cxy**).

**Section** The name of the section which has been removed.

**File** The name of the input object file where the section is removed from.

**Library** The name of the library where the object file is part of.

**Symbol** The symbols that were present in the section.

**Reason** The reason why the section has been removed. This can be because the section is unreferenced or duplicated.

# Chapter 7. Linker Script Language (LSL)

To make full use of the linker, you can write a script with information about the architecture of the target processor and locating information. The language for the script is called the *Linker Script Language (LSL)*. This chapter first describes the structure of an LSL file. The next section contains a summary of the LSL syntax. In the remaining sections, the semantics of the Linker Script Language is explained.

The TASKING linker is a target independent linker/locator that can simultaneously link and locate all programs for all cores available on a target board. The target board may be of arbitrary complexity. A simple target board may contain one standard processor with some external memory that executes one task. A complex target board may contain multiple standard processors and DSPs combined with configurable IP-cores loaded in an FPGA. Each core may execute a different program, and external memory may be shared by multiple cores.

LSL serves two purposes. First it enables you to specify the characteristics (that are of interest to the linker) of your specific target board and of the cores installed on the board. Second it enables you to specify how sections should be located in memory.

## 7.1. Structure of a Linker Script File

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

### The architecture definition (required)

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

This specification is normally written by Altium. Altium supplies LSL files in the `include.lsl` directory. The architecture definition of the LSL file should not be changed by you unless you also modify the core's hardware architecture. If the LSL file describes a multi-core system an architecture definition must be available for each different type of core.

See Section 7.4, *Semantics of the Architecture Definition* for detailed descriptions of LSL in the architecture definition.

### The derivative definition

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

Altium provides LSL descriptions of supported derivatives. When you build an ASIC or use a derivative that is not (yet) supported by the TASKING tools, you may have to write a derivative definition.

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

See Section 7.5, *Semantics of the Derivative Definition* for a detailed description of LSL in the derivative definition.

## The processor definition

The *processor definition* describes an instance of a derivative. Typically the processor definition instantiates one derivative only (single-core processor). A processor that contains multiple cores having the same (homogeneous) or different (heterogeneous) architecture can also be described by instantiating multiple derivatives of the same or different types in separate processor definitions.

See Section 7.6, *Semantics of the Board Specification* for a detailed description of LSL in the processor definition.

## The memory and bus definitions (optional)

Memory and bus definitions are used within the context of a derivative definition to specify internal memory and on-chip buses. In the context of a board specification the memory and bus definitions are used to define external (off-chip) memory and buses. Given the above definitions the linker can convert a logical address into an offset into an on-chip or off-chip memory device.

See Section 7.6.3, *Defining External Memory and Buses*, for more information on how to specify the external physical memory layout. *Internal* memory for a processor should be defined in the derivative definition for that processor.

## The board specification

The processor definition and memory and bus definitions together form a *board specification*. LSL provides language constructs to easily describe single-core and heterogeneous or homogeneous multi-core systems. The board specification describes all characteristics of your target board's system buses, memory devices, I/O sub-systems, and cores that are of interest to the linker. Based on the information provided in the board specification the linker can for each core:

- convert a logical address to an offset within a memory device
- locate sections in physical memory
- maintain an overall view of the used and free physical memory within the whole system while locating

## The section layout definition (optional)

The optional section layout definition enables you to exactly control where input sections are located. Features are provided such as: the ability to place sections at a given load-address or run-time address, to place sections in a given order, and to overlay code and/or data sections.

Which object files (sections) constitute the task that will run on a given core is specified on the command line when you invoke the linker. The linker will link and locate all sections of all tasks simultaneously. From the section layout definition the linker can deduce where a given section may be located in memory, form the board specification the linker can deduce which physical memory is (still) available while locating the section.

See Section 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

```
architecture architecture_name
{
    // Specification core architecture
}

derivative derivative_name
{
    // Derivative definition
}

processor processor_name
{
    // Processor definition
}

memory and/or bus definitions

section_layout space_name
{
    // section placement statements
}
```

## 7.2. Syntax of the Linker Script Language

This section describes what the LSL language looks like. An LSL document is stored as a file coded in UTF-8 with extension `.lsl`. Before processing an LSL file, the linker preprocesses it using a standard C preprocessor. Following this, the linker interprets the LSL file using a scanner and parser. Finally, the linker uses the information found in the LSL file to guide the locating process.

### 7.2.1. Preprocessing

When the linker loads an LSL file, the linker processes it with a C-style prepocessor. As such, it strips C and C++ comments. You can use the standard ISO C preprocessor directives, such as `#include`, `#define`, `#if/#else/#endif`, `#error`.

For example:

```
#include "arch.lsl"
```

Preprocess and include the file `arch.lsl` at this point in the LSL file.

## 7.2.2. Lexical Syntax

The following lexicon is used to describe the syntax of the Linker Script Language:

| | | |
|---|---|---|
| *A ::= B* | = | *A* is defined as *B* |
| *A ::= B C* | = | *A* is defined as *B* and *C*; *B* is followed by *C* |
| *A ::= B \| C* | = | *A* is defined as *B* or *C* |
| <*B*>$^{0\|1}$ | = | zero or one occurrence of *B* |
| <*B*>$^{>=0}$ | = | zero of more occurrences of *B* |
| <*B*>$^{>=1}$ | = | one of more occurrences of *B* |
| *IDENTIFIER* | = | a character sequence starting with 'a'-'z', 'A'-'Z' or '_'. Following characters may also be digits and dots '.' |
| *STRING* | = | sequence of characters not starting with \n, \r or \t |
| *DQSTRING* | = | `" STRING "` (double quoted string) |
| *OCT_NUM* | = | octal number, starting with a zero (`06`, `045`) |
| *DEC_NUM* | = | decimal number, not starting with a zero (`14`, `1024`) |
| *HEX_NUM* | = | hexadecimal number, starting with '0x' (`0x0023`, `0xFF00`) |

*OCT_NUM*, *DEC_NUM* and *HEX_NUM* can be followed by a **k** (kilo), **M** (mega), or **G** (giga).

Characters in **bold** are characters that occur literally. Words in *italics* are higher order terms that are defined in the same or in one of the other sections.

To write comments in LSL file, you can use the C style '`/* */`' or C++ style '`//`'.

## 7.2.3. Identifiers and Tags

```
arch_name          ::= IDENTIFIER
bus_name           ::= IDENTIFIER
core_name          ::= IDENTIFIER
derivative_name    ::= IDENTIFIER
file_name          ::= DQSTRING
group_name         ::= IDENTIFIER
heap_name          ::= section_name
map_name           ::= IDENTIFIER
mem_name           ::= IDENTIFIER
proc_name          ::= IDENTIFIER
section_name       ::= DQSTRING
space_name         ::= IDENTIFIER
stack_name         ::= section_name
symbol_name        ::= DQSTRING
```

```
tag_attr          ::= (tag<,tag>>=0)
tag               ::= tag = DQSTRING
```

A tag is an arbitrary text that can be added to a statement.

### 7.2.4. Expressions

The expressions and operators in this section work the same as in ISO C.

```
number            ::= OCT_NUM
                    | DEC_NUM
                    | HEX_NUM

expr              ::= number
                    | symbol_name
                    | unary_op expr
                    | expr binary_op expr
                    | expr ? expr : expr
                    | ( expr )
                    | function_call

unary_op          ::= !    // logical NOT
                    | ~    // bitwise complement
                    | -    // negative value

binary_op         ::= ^    // exclusive OR
                    | *    // multiplication
                    | /    // division
                    | %    // modulus
                    | +    // addition
                    | -    // subtraction
                    | >>   // right shift
                    | <<   // left shift
                    | ==   // equal to
                    | !=   // not equal to
                    | >    // greater than
                    | <    // less than
                    | >=   // greater than or equal to
                    | <=   // less than or equal to
                    | &    // bitwise AND
                    | |    // bitwise OR
                    | &&   // logical AND
                    | ||   // logical OR
```

### 7.2.5. Built-in Functions

```
function_call     ::= absolute ( expr )
                    | addressof ( addr_id )
                    | exists ( section_name )
                    | max ( expr , expr )
```

```
                      | min ( expr , expr )
                      | sizeof ( size_id )

addr_id             ::= sect : section_name
                      | group : group_name

size_id             ::= sect : section_name
                      | group : group_name
                      | mem : mem_name
```

- Every space, bus, memory, section or group you refer to, must be defined in the LSL file.
- The `addressof()` and `sizeof()` functions with the **`group`** or **`sect`** argument can only be used in the right hand side of an assignment. The `sizeof()` function with the **`mem`** argument can be used anywhere in section layouts.

You can use the following built-in functions in expressions. All functions return a numerical value. This value is a 64-bit signed integer.

### absolute()

```
int absolute( expr )
```

Converts the value of *expr* to a positive integer.

```
absolute( "labelA"-"labelB" )
```

### addressof()

```
int addressof( addr_id )
```

Returns the address of *addr_id*, which is a named section or group. To get the offset of the section with the name `asect`:

```
addressof( sect: "asect")
```

> This function only works in assignments.

### exists()

```
int exists( section_name )
```

The function returns 1 if the section *section_name* exists in one or more object file, 0 otherwise. If the section is not present in input object files, but generated from LSL, the result of this function is undefined.

To check whether the section `mysection` exists in one of the object files that is specified to the linker:

```
exists( "mysection" )
```

### max()

`int max(` *`expr`*`,` *`expr`* `)`

Returns the value of the expression that has the largest value. To get the highest value of two symbols:

```
max( "sym1" , "sym2")
```

### min()

`int min(` *`expr`*`,` *`expr`* `)`

Returns the value of the expression hat has the smallest value. To get the lowest value of two symbols:

```
min( "sym1" , "sym2")
```

### sizeof()

`int sizeof(` *`size_id`* `)`

Returns the size of the object (group, section or memory) the identifier refers to. To get the size of the section "asection":

```
sizeof( sect: "asection" )
```

The **`group`** and **`sect`** arguments only works in assignments. The **`mem`** argument can be used anywhere in section layouts.

## 7.2.6. LSL Definitions in the Linker Script File

*description* ::= <*definition*>$^{>=1}$

*definition* ::= *architecture_definition*
| *derivative_definition*
| *board_spec*
| *section_definition*
| *section_setup*

- At least one *architecture_definition* must be present in the LSL file.

## 7.2.7. Memory and Bus Definitions

*mem_def* ::= **memory** *mem_name* <*tag_attr*>$^{0|1}$ **{** <*mem_descr* ;>$^{>=0}$ **}**

- A *mem_def* defines a memory with the *mem_name* as a unique name.

*mem_descr* ::= **type =** <**reserved**>$^{0|1}$ *mem_type*
| **mau =** *expr*
| **size =** *expr*
| **speed =** *number*

```
                        | priority = number
                        | exec_priority = number
                        | fill <= fill_values>0|1
                        | write_unit = expr
                        | mapping
```

- A *mem_def* contains exactly one **type** statement.
- A *mem_def* contains exactly one **mau** statement (non-zero size).
- A *mem_def* contains exactly one **size** statement.
- A *mem_def* contains zero or one **priority** (or **speed**) statement (if absent, the default value is 1).
- A *mem_def* contains zero or one **exec_priority** statement.
- A *mem_def* contains zero or one **fill** statement.
- A *mem_def* contains zero or one **write_unit** statement.
- A *mem_def* contains at least one *mapping*

```
mem_type              ::= rom         // attrs = rx
                        | ram         // attrs = rw
                        | nvram       // attrs = rwx
                        | blockram

fill_values           ::= expr
                        | [ expr <, expr>>=0 ]

bus_def               ::= bus bus_name {  <bus_descr ;>>=0 }
```

- A *bus_def* statement defines a bus with the given *bus_name* as a unique name within a core architecture.

```
bus_descr             ::= mau = expr
                        | width = expr  // bus width, nr
                        |               // of data bits
                        | mapping       // legal destination
                                        // 'bus' only
```

- The **mau** and **width** statements appear exactly once in a *bus_descr*. The default value for **width** is the **mau** size.
- The bus width must be an integer times the bus MAU size.
- The MAU size must be non-zero.
- A bus can only have a *mapping* on a destination bus (through **dest = bus:** ).

```
mapping               ::= map <map_name>0|1 ( map_descr <, map_descr>>=0 )
```

```
map_descr          ::= dest = destination
                     | dest_dbits = range
                     | dest_offset = expr
                     | size = expr
                     | src_dbits = range
                     | src_offset = expr
                     | reserved
                     | priority = number
                     | exec_priority = number
                     | tag
```

- A *map_descr* requires at least the **size** and **dest** statements.
- A *map_descr* contains zero or one **priority** statement (if absent, the default value is 0).
- A *map_descr* contains zero or one **exec_priority** statement.
- Each *map_descr* can occur only once.
- You can define multiple mappings from a single source.
- Overlap between source ranges or destination ranges is not allowed.
- If the **src_dbits** or **dest_dbits** statement is not present, its value defaults to the **width** value if the source/destination is a bus, and to the **mau** size otherwise.
- The **reserved** statement is allowed only in mappings defined for a memory.

```
destination        ::= space : space_name
                     | bus : <proc_name |
                                core_name :>^{0|1} bus_name
```

- A *space_name* refers to a defined address space.
- A *proc_name* refers to a defined processor.
- A *core_name* refers to a defined core.
- A *bus_name* refers to a defined bus.
- The following mappings are allowed (source to destination)
  - space => space
  - space => bus
  - bus => bus
  - memory => bus

```
range              ::= expr .. expr
```

- With address ranges, the end address is not part of the range.

### 7.2.8. Architecture Definition

```
architecture_definition
                 ::= architecture arch_name
                     <( parameter_list )>0|1
                     <extends arch_name
                             <( argument_list )>0|1 >0|1
                     { <arch_spec>>=0 }
```

- An *architecture_definition* defines a core architecture with the given *arch_name* as a unique name.
- At least one *space_def* and at least one *bus_def* have to be present in an *architecture_definition*.
- An *architecture_definition* that uses the **extends** construct defines an architecture that inherits all elements of the architecture defined by the second *arch_name*. The parent architecture must be defined in the LSL file as well.

```
parameter_list   ::= parameter <, parameter>>=0

parameter        ::= IDENTIFIER <= expr>0|1

argument_list    ::= expr <, expr>>=0

arch_spec        ::= bus_def
                   | space_def
                   | endianness_def

space_def        ::= space space_name <tag_attr>0|1 { <space_descr;>>=0 }
```

- A *space_def* defines an address space with the given *space_name* as a unique name within an architecture.

```
space_descr      ::= space_property ;
                   | section_definition  //no space ref
                   | reserved_range

space_property   ::= id = number // as used in object
                   | mau = expr
                   | align = expr
                   | page_size = expr <[ range ] <| [ range ]>>=0>0|1
                   | page
                   | direction = direction
                   | stack_def
                   | heap_def
                   | copy_table_def
                   | start_address
                   | mapping
```

- A *space_def* contains exactly one **id** and one **mau** statement.

- A *space_def* contains at most one **align** statement.
- A *space_def* contains at most one **page_size** statement.
- A *space_def* contains at least one *mapping*.

```
stack_def         ::= stack stack_name ( stack_heap_descr
                            <, stack_heap_descr >>=0 )
```

- A *stack_def* defines a stack with the *stack_name* as a unique name.

```
heap_def          ::= heap heap_name ( stack_heap_descr
                            <, stack_heap_descr >>=0 )
```

- A *heap_def* defines a heap with the *heap_name* as a unique name.

```
stack_heap_descr  ::= min_size = expr
                    | grows = direction
                    | align = expr
                    | fixed
                    | tag
```

- 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 heap grow **low-to-high**.

```
copy_table_def    ::= copytable <(  copy_table_descr
                            <, copy_table_descr >>=0 )>0|1
```

- A *space_def* contains at most one **copytable** statement.
- Exactly one copy table must be defined in one of the spaces.

```
copy_table_descr  ::= align = expr
                    | copy_unit = expr
                    | dest <space_name>0|1 = space_name
                    | page
                    | tag
```

- The **copy_unit** is defined by the size in MAUs in which the startup code moves data.
- The **dest** statement is only required when the startup code initializes memory used by another processor that has no access to ROM.
- A *space_name* refers to a defined address space.

```
start_addr        ::= start_address ( start_addr_descr
                               <, start_addr_descr>>=0 )
```

*start_addr_descr* ::= **run_addr =** *expr*
| **symbol =** *symbol_name*

- A *symbol_name* refers to the section that contains the startup code.

*reserved_range* ::= **reserved** <*tag_attr*>$^{0|1}$ *expr* **..** *expr* **;**

- The end address is not part of the range.

*endianness_def* ::= **endianness {** <*endianness_type*;>$^{>=1}$ **}**

*endianness_type* ::= **big**
| **little**

## 7.2.9. Derivative Definition

*derivative_definition*
::= **derivative** *derivative_name*
<**(** *parameter_list* **)**>$^{0|1}$
<**extends** *derivative_name*
<**(** *argument_list* **)**>$^{0|1}$ >$^{0|1}$
**{** <*derivative_spec*>$^{>=0}$ **}**

- A *derivative_definition* defines a derivative with the given *derivative_name* as a unique name.

*derivative_spec* ::= *core_def*
| *bus_def*
| *mem_def*
| *section_definition* // no processor name
| *section_setup*

*core_def* ::= **core** *core_name* **{** <*core_descr* **;**>$^{>=0}$ **}**

- A *core_def* defines a core with the given *core_name* as a unique name.

- At least one *core_def* must be present in a *derivative_definition*.

*core_descr* ::= **architecture =** *arch_name*
<**(** *argument_list* **)**>$^{0|1}$
| **copytable_space** <*core_name* **:**>$^{0|1}$ *space_name*
| **endianness = (** *endianness_type*
<**,** *endianness_type*>$^{>=0}$ **)**
| **import** *core_name*
| **space_id_offset =** *number*

- An *arch_name* refers to a defined core architecture.

- Exactly one **architecture** statement must be present in a *core_def*.

- Exactly one **copytable_space** statement must be present in a *core_def*, or in exactly one space in that core, a **copytable** statement must be present.

## 7.2.10. Processor Definition and Board Specification

```
board_spec          ::= proc_def
                      | bus_def
                      | mem_def

proc_def            ::= processor proc_name
                        { proc_descr ; }

proc_descr          ::= derivative = derivative_name
                        <( argument_list )>0|1
```

- A *proc_def* defines a processor with the *proc_name* as a unique name.
- If you do not explicitly define a processor for a derivative in an LSL file, the linker defines a processor with the same name as that derivative.
- A *derivative_name* refers to a defined derivative.
- A *proc_def* contains exactly one **derivative** statement.

## 7.2.11. Section Layout Definition and Section Setup

```
section_definition ::= section_layout <space_ref>0|1
                       <( space_layout_properties )>0|1
                       { <section_statement>>=0 }
```

- A section definition inside a space definition does not have a *space_ref*.
- All global section definitions have a *space_ref*.

```
space_ref           ::= <proc_name>0|1 : <core_name>0|1
                        : space_name <| space_name>>=0
```

- If more than one processor is present, the *proc_name* must be given for a global section layout.
- If the section layout refers to a processor that has more than one core, the *core_name* must be given in the *space_ref*.
- A *proc_name* refers to a defined processor.
- A *core_name* refers to a defined core.
- A *space_name* refers to a defined address space.

```
space_layout_properties
                    ::= space_layout_property <, space_layout_property >>=0

space_layout_property
                    ::= locate_direction
                      | tag
```

*locate_direction* ::= **direction =** *direction*

*direction* ::= **low_to_high**
| **high_to_low**

- A section layout contains at most one **direction** statement.

- If you do not specify the **direction** statement, the locate direction of the section layout is **low-to-high**.

*section_statement*
::= *simple_section_statement* **;**
| *aggregate_section_statement*

*simple_section_statement*
::= *assignment*
| *select_section_statement*
| *special_section_statement*
| *memcopy_statement*

*assignment* ::= *symbol_name assign_op expr*

*assign_op* ::= **=**
| **:=**

*select_section_statement*
::= **select** <**ref_tree**>$^{0|1}$ <*section_name*>$^{0|1}$
<*section_selections*>$^{0|1}$

- Either a *section_name* or at least one *section_selection* must be defined.

*section_selections*
::= **(** *section_selection*
<**,** *section_selection*>$^{>=0}$ **)**

*section_selection*
::= **attributes =** < <**+**|**-**> *attribute*>$^{>0}$
| *tag*

- ***+****attribute* means: select all sections that have this attribute.

- **-***attribute* means: select all sections that do not have this attribute.

*special_section_statement*
::= **heap** *heap_name* <*stack_heap_mods*>$^{0|1}$
| **stack** *stack_name* <*stack_heap_mods*>$^{0|1}$
| **copytable**
| **reserved** *section_name* <*reserved_specs*>$^{0|1}$

- Special sections cannot be selected in load-time groups.

*stack_heap_mods* ::= **(** *stack_heap_mod* <**,** *stack_heap_mod*>$^{>=0}$ **)**

```
stack_heap_mod     ::= size = expr
                     | tag

reserved_specs     ::= ( reserved_spec <, reserved_spec>^{>=0} )

reserved_spec      ::= attributes
                     | fill_spec
                     | size = expr
                     | alloc_allowed = absolute | ranged
```

- If a **reserved** section has attributes **r**, **rw**, **x**, **rx** or **rwx**, and no fill pattern is defined, the section is filled with zeros. If no attributes are set, the section is created as a scratch section (attributes **ws**, no image).

```
memcopy_statement
                   ::= memcopy section_name
                       ( memcopy_spec <, memcopy_spec>^{0|1} )

memcopy_spec       ::= memory = memory_reference
                     | fill_spec
```

- A **memcopy** statement must contain exactly one **memory** statement.

- A **memcopy** statement can contain at most one *fill_spec*.

```
fill_spec          ::= fill = fill_values

fill_values        ::= expr
                     | [ expr <, expr>^{>=0} ]

aggregate_section_statement
                   ::= { <section_statement>^{>=0} }
                     | group_descr
                     | if_statement
                     | section_creation_statement

group_descr        ::= group <group_name>^{0|1} <( group_specs )>^{0|1}
                             section_statement
```

- For every group with a name, the linker defines a label.

- No two groups for address spaces of a core can have the same *group_name*.

```
group_specs        ::= group_spec <, group_spec >^{>=0}

group_spec         ::= group_alignment
                     | attributes
                     | copy
                     | nocopy
                     | group_load_address
                     | fill <= fill_values>^{0|1}
                     | group_page
                     | group_run_address
```

```
                         | group_type
                         | allow_cross_references
                         | priority = number
                         | tag
```

- The **`allow-cross-references`** property is only allowed for *overlay* groups.
- Sub groups inherit all properties from a parent group.

```
group_alignment   ::= align = expr

attributes        ::= attributes = <attribute>>=1

attribute         ::= r    // readable sections
                    | w    // writable sections
                    | x    // executable code sections
                    | i    // initialized sections
                    | s    // scratch sections
                    | b    // blanked (cleared) sections
                    | p    // protected sections

group_load_address
                  ::= load_addr <= load_or_run_addr>0|1

group_page        ::= page <= expr>0|1
                    | page_size = expr <[ range ] <| [ range ]>>=0>0|1

group_run_address ::= run_addr <= load_or_run_addr>0|1

group_type        ::= clustered
                    | contiguous
                    | ordered
                    | overlay
```

- For *non-contiguous* groups, you can only specify `group_alignment` and `attributes`.
- The **`overlay`** keyword also sets the **`contiguous`** property.
- The **`clustered`** property cannot be set together with **`contiguous`** or **`ordered`** on a single group.

```
load_or_run_addr  ::= addr_absolute
                    | addr_range <| addr_range>>=0

addr_absolute     ::= expr
                    | memory_reference [ expr ]
```

- An absolute address can only be set on *ordered* groups.

```
addr_range        ::= [ expr .. expr ]
                    | memory_reference
                    | memory_reference [ expr .. expr ]
```

- The parent of a group with an *addr_range* or **page** restriction cannot be **ordered**, **contiguous** or **clustered**.
- The end address is not part of the range.

*memory_reference* ::= **mem :** <*proc_name* **:**>$^{0|1}$ *mem_name* <**/** *map_name*>$^{0|1}$

- A *proc_name* refers to a defined processor.
- A *mem_name* refers to a defined memory.
- A *map_name* refers to a defined memory mapping.

*if_statement* ::= **if (** *expr* **)** *section_statement*
<**else** *section_statement*>$^{0|1}$

*section_creation_statement*
::= **section** *section_name* **(** *section_specs* **)**
**{** <*section_statement2*>$^{>=0}$ **}**

*section_specs* ::= *section_spec* <**,** *section_spec* >$^{>=0}$

*section_spec* ::= *attributes*
| *fill_spec*
| **size =** *expr*
| **blocksize =** *expr*
| **overflow =** *section_name*
| *tag*

*section_statement2*
::= *select_section_statement* **;**
| *group_descr2*
| **{** <*section_statement2*>$^{>=0}$ **}**

*group_descr2* ::= **group** <*group_name*>$^{0|1}$
**(** *group_specs2* **)**
*section_statement2*

*group_specs2* ::= *group_spec2* <**,** *group_spec2* >$^{>=0}$

*group_spec2* ::= *group_alignment*
| *attributes*
| **load_addr**
| *tag*

*section_setup* ::= **section_setup** *space_ref* <*tag_attr*>$^{0|1}$
**{** <*section_setup_item*>$^{>=0}$ **}**

*section_setup_item*
::= *reserved_range*
| *stack_def* **;**
| *heap_def* **;**

## 7.3. Expression Evaluation

Only *constant* expressions are allowed, including sizes, but not addresses, of sections in object files.

All expressions are evaluated with 64-bit precision integer arithmetic. The result of an expression can be absolute or relocatable. A symbol you assign is created as an absolute symbol.

## 7.4. Semantics of the Architecture Definition

### Keywords in the architecture definition

```
architecture
   extends
endianness          big  little
bus
   mau
   width
   map
space
   id
   mau
   align
   page_size
   page
   direction        low_to_high  high_to_low
   stack
      min_size
      grows         low_to_high  high_to_low
      align
      fixed
   heap
      min_size
      grows         low_to_high  high_to_low
      align
      fixed
   copytable
      align
      copy_unit
      dest
      page
  reserved
   start_address
      run_addr
      symbol
   map

   map
      dest          bus  space
      dest_dbits
      dest_offset
      size
      src_dbits
      src_offset
      priority
      exec_priority
```

### 7.4.1. Defining an Architecture

With the keyword **architecture** you define an architecture and assign a unique name to it. The name is used to refer to it at other places in the LSL file:

```
architecture name
{
     definitions
}
```

If you are defining multiple core architectures that show great resemblance, you can define the common features in a parent core architecture and extend this with a child core architecture that contains specific features. The child inherits all features of the parent. With the keyword extends you create a child core architecture:

```
architecture name_child_arch extends name_parent_arch
{
     definitions
}
```

A core architecture can have any number of parameters. These are identifiers which get values assigned on instantiation or extension of the architecture. You can use them in any expression within the core architecture. Parameters can have default values, which are used when the core architecture is instantiated with less arguments than there are parameters defined for it. When you extend a core architecture you can pass arguments to the parent architecture. Arguments are expressions that set the value of the parameters of the sub-architecture.

```
architecture name_child_arch (parm1,parm2=1)
             extends name_parent_arch (arguments)
{
     definitions
}
```

### 7.4.2. Defining Internal Buses

With the **bus** keyword you define a bus (the combination of data and corresponding address bus). The bus name is used to identify a bus and does not conflict with other identifiers. Bus descriptions in an architecture definition or derivative definition define *internal* buses. Some internal buses are used to communicate with the components outside the core or processor. Such buses on a processor have physical pins reserved for the number of bits specified with the **width** statements.

- The **mau** field specifies the MAU size (Minimum Addressable Unit) of the data bus. This field is required.

- The **width** field specifies the width (number of address lines) of the data bus. The default value is the MAU size.
- The **map** keyword specifies how this bus maps onto another bus (if so). Mappings are described in Section 7.4.4, *Mappings*.

```
bus bus_name
{
    mau = 8;
    width = 8;
    map ( map_description );
}
```

## 7.4.3. Defining Address Spaces

With the **space** keyword you define a logical address space. The space name is used to identify the address space and does not conflict with other identifiers.

- The **id** field defines how the addressing space is identified in object files. In general, each address space has a unique ID. The linker locates sections with a certain ID in the address space with the same ID. This field is required.
- The **mau** field specifies the MAU size (Minimum Addressable Unit) of the space. This field is required.
- The **align** value must be a power of two. The linker uses this value to compute the start addresses when sections are concatenated. An align value of *n* means that objects in the address space have to be aligned on *n* MAUs.
- The **page_size** field sets the page alignment and page size in MAUs for the address space. It must be a power of 2. The default value is 1. If one or more page ranges are supplied the supplied value only sets the page alignment. The ranges specify the available space in each page, as offsets to the page start, which is aligned at the page alignment.

  See also the **page** keyword in subsection Locating a group in Section 7.8.2, *Creating and Locating Groups of Sections*.
- With the optional **direction** field you can specify how all sections in this space should be located. This can be either from **low_to_high** addresses (this is the default) or from **high_to_low** addresses.
- The **map** keyword specifies how this address space maps onto an internal bus or onto another address space. Mappings are described in Section 7.4.4, *Mappings*.

### Stacks and heaps

- The **stack** keyword defines a stack in the address space and assigns a name to it. The architecture definition must contain at least one stack definition. Each stack of a core architecture must have a unique name. See also the **stack** keyword in Section 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*.

Stacks and heaps are only generated by the linker if the corresponding linker labels are referenced in the object files.

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. You must define exactly one copy table in one of the address spaces (for a core).

  Optionally you can specify an alignment for the copy table with the argument **`align`**. This alignment must be equal or larger than the alignment that you specify for the address space itself. If smaller, the alignment for the address space is used.

  The **`copy_unit`** argument specifies the size in MAUs of information chunks that are copied. If you do not specify the copy unit, the MAU size of the address space itself is used.

  The **`dest`** argument specifies the destination address space that the code uses for the copy table. The linker uses this information to generate the correct addresses in the copy table. The memory into where the sections must be copied at run-time, must be accessible from this destination space.

  Sections generated for the copy table may get a page restriction with the address space's page size, by adding the **`page`** argument.

## Reserved address ranges

- The **`reserved`** keyword specifies to reserve a part of an address space even if not all of the range is covered by memory. See also the **`reserved`** keyword in Section 7.8.3, *Creating or Modifying Special Sections*.

## Start address

- The **`start_address`** keyword specifies the start address for the position where the startup code is located. When a processor is reset, it initializes its program counter to a certain start address, sometimes called the reset vector. In the architecture definition, you must specify this start address in the correct address space in combination with the name of the label in the application code which must be located here.

The **run_addr** argument specifies the start address (reset vector). If the core starts executing using an entry from a vector table, and directly jumps to the start label, you should omit this argument.

The **symbol** argument specifies the name of the label in the application code that should be located at the specified start address. The **symbol** argument is required. The linker will resolve the start symbol and use its value after locating for the start address field in IEEE-695 files and Intel Hex files. If you also specified the **run_addr** argument, the start symbol (label) must point to a section. The linker locates this section such that the start symbol ends up on the start address.

```
space space_name
{
     id = 1;
     mau = 8;
     align = 8;
     page_size = 1;
     stack name (min_size = 1k, grows = low_to_high);
     reserved start_address .. end_address;
     start_address ( run_addr = 0x0000,
                     symbol = "start_label" )
     map ( map_description );
}
```

## 7.4.4. Mappings

You can use a mapping when you define a space, bus or memory. With the **map** field you specify how addresses from the source (space, bus or memory) are translated to addresses of a destination (space, bus). The following mappings are possible:

- space => space
- space => bus
- bus => bus
- memory => bus

With a mapping you specify a range of source addresses you want to map (specified by a source offset and a size), the destination to which you want to map them (a bus or another address space), and the offset address in the destination.

- The **dest** argument specifies the destination. This can be a **bus** or another address **space** (only for a space to space mapping). This argument is required.
- The **src_offset** argument specifies the offset of the source addresses. In combination with size, this specifies the range of address that are mapped. By default the source offset is 0x0000.
- The **size** argument specifies the number of addresses that are mapped. This argument is required.

- The **`dest_offset`** argument specifies the position in the destination to which the specified range of addresses is mapped. By default the destination offset is 0x0000.

If you are mapping a bus to another bus, the number of data lines of each bus may differ. In this case you have to specify a range of source data lines you want to map (**`src_dbits =`** *`begin..end`*) and the range of destination data lines you want to map them to (**`dest_dbits =`** *`first..last`*).

- The **`src_dbits`** argument specifies a range of data lines of the source bus. By default all data lines are mapped.

- The **`dest_dbits`** argument specifies a range of data lines of the destination bus. By default, all data lines from the source bus are mapped on the data lines of the destination bus (starting with line 0).

If you define a memory and the memory mapping must not be used by default when locating sections in address spaces, you can specify the **`reserved`** argument. This marks all address space areas that the mapping points to as reserved. If a section has an absolute or address range restriction, the reservation is lifted and the section may be located at these locations. This feature is only useful when more than one mapping is available for a range of memory addresses, otherwise the **`memory`** keyword with the same name would be used.

For example:

```
memory xrom
{
    mau = 8;
    size = 1M;
    type = rom;
    map     cached (dest=bus:spe:fpi_bus, dest_offset=0x80000000,
                    size=1M);
    map not_cached (dest=bus:spe:fpi_bus, dest_offset=0xa0000000,
                    size=1M, reserved);
}
```

## Mapping priority

If you define a memory you can set a locate priority on a mapping with the keywords **`priority`** and **`exec_priority`**. The values of these priorities are relative which means they add to the priority of memories. Whereas a priority set on the memory applies to all address space areas reachable through any mapping of the memory, a priority set on a mapping only applies to address space areas reachable through the mapping. The memory mapping with the highest priority is considered first when locating. To set only a priority for non-executable (data) sections, add a **`priority`** keyword with the desired value and an **`exec_priority`** set to zero. To set only a priority for executable (code) sections, simply set an **`exec_priority`** keyword to the desired value.

The default for a mapping **`priority`** is zero, while the default for **`exec_priority`** is the same as the specified **`priority`**. If you specify a value for **`priority`** in LSL it must be greater than zero. A value for **`exec_priority`** must be greater or equal to zero.

For more information about priority values see the description of the memory **`priority`** keyword.

```
memory dspram
{
    mau = 8;
    size = 112k;
    type = ram;
    map (dest=bus:mycore_1:fpi_bus, dest_offset=0xd0000000,
              size=112k, priority=8, exec_priority=0);
    map (dest=bus:sri, dest_offset=0x70000000,
              size=112k);
}
```

## From space to space

If you map an address space to another address space (nesting), you can do this by mapping the subspace to the containing larger space. In this example a small space of 64 kB is mapped on a large space of 16 MB.

```
space small
{
   id = 2;
   mau = 4;
   map (src_offset = 0, dest_offset = 0,
        dest = space : large, size = 64k);
}
```

## From space to bus

All spaces that are not mapped to another space must map to a bus in the architecture:

```
space large
{
   id = 1;
   mau = 4;
   map (src_offset = 0, dest_offset = 0,
        dest = bus:bus_name, size = 16M );
}
```

## From bus to bus

The next example maps an external bus called `e_bus` to an internal bus called `i_bus`. This internal bus resides on a core called `mycore`. The source bus has 16 data lines whereas the destination bus has only 8 data lines. Therefore, the keywords **`src_dbits`** and **`dest_dbits`** specify which source data lines are mapped on which destination data lines.

```
architecture mycore
{
    bus i_bus
    {
       mau = 4;
    }
    space i_space
    {
       map (dest=bus:i_bus, size=256);
    }
}

bus e_bus
{
   mau = 16;
   width = 16;
   map (dest = bus:mycore:i_bus, src_dbits = 0..7, dest_dbits = 0..7 )
}
```

It is not possible to map an internal bus to an external bus.

# 7.5. Semantics of the Derivative Definition

## Keywords in the derivative definition

```
derivative
   extends
core
   architecture
   import
   space_id_offset
   copytable_space
bus
   mau
   width
   map
memory
   type             reserved rom  ram  nvram  blockram
   mau
   size
   speed
   priority
   exec_priority
   fill
   write_unit
   map
section_layout
section_setup

   map
      dest          bus  space
      dest_dbits
      dest_offset
      size
      src_dbits
      src_offset
      priority
      exec_priority
      reserved
```

### 7.5.1. Defining a Derivative

With the keyword **derivative** you define a derivative and assign a unique name to it. The name is used to refer to it at other places in the LSL file:

```
derivative name
{
     definitions
}
```

If you are defining multiple derivatives that show great resemblance, you can define the common features in a parent derivative and extend this with a child derivative that contains specific features. The child inherits all features of the parent (cores and memories). With the keyword **extends** you create a child derivative:

```
derivative name_child_deriv extends name_parent_deriv
{
     definitions
}
```

As with a core architecture, a derivative can have any number of parameters. These are identifiers which get values assigned on instantiation or extension of the derivative. You can use them in any expression within the derivative definition.

```
derivative name_child_deriv (parm1,parm2=1)
           extends name_parent_deriv (arguments)
{
     definitions
}
```

### 7.5.2. Instantiating Core Architectures

With the keyword **core** you instantiate a core architecture in a derivative.

- With the keyword **architecture** you tell the linker that the given core has a certain architecture. The architecture name refers to an existing architecture definition in the same LSL file.

  For example, if you have two cores (called `mycore_1` and `mycore_2`) that have the same architecture (called `mycorearch`), you must instantiate both cores as follows:

  ```
  core mycore_1
  {
       architecture = mycorearch;
  }
  ```

```
core mycore_2
{
     architecture = mycorearch;
}
```

If the architecture definition has parameters you must specify the arguments that correspond with the parameters. For example `mycorearch1` expects two parameters which are used in the architecture definition:

```
core mycore
{
     architecture = mycorearch1 (1,2);
}
```

- With the keyword **`import`** you can combine multiple cores with the same architecture into a single link task. The imported cores share a single symbol namespace.
- The address spaces in each imported core must have a unique ID in the link task. With the keyword **`space_id_offset`** you specify for each imported core that the space IDs of the imported core start at a specific offset.
- With the keyword **`copytable_space`** you can specify that writable sections for a core must be initialized by using the copy table of a different core.

```
core mycore_1
{
    architecture = mycorearch;
    space_id_offset = 100; // add 100 to all space IDs in
                           // the architecture definition
    copytable_space = mycore:myspace; // use copytable from core mycore
}
core mycore_2
{
    architecture = mycorearch;
    space_id_offset = 200; // add 200 to all space IDs in
                           // the architecture definition
    copytable_space = mycore:myspace; // use copytable from core mycore
}

core mycore
{
    architecture = mycorearch;
    import mycore_1; // add all address spaces of mycore_1 for linking
    import mycore_2; // add all address spaces of mycore_2 for linking
}
```

### 7.5.3. Defining Internal Memory and Buses

With the keyword **memory** you define physical memory that is present on the target board. The memory name is used to identify the memory and does not conflict with other identifiers. It is common to define internal memory (on-chip) in the derivative definition. External memory (off-chip memory) is usually defined in the board specification (See Section 7.6.3, *Defining External Memory and Buses*).

- The **type** field specifies a memory type:
  - **rom**: read-only memory - it can only be written at load-time
  - **ram**: random access volatile writable memory - writing at run-time is possible while writing at load-time has no use since the data is not retained after a power-down
  - **nvram**: non volatile ram - writing is possible both at load-time and run-time
  - **blockram**: writing is possible both at load-time and run-time. Changes are applied in RAM, so after a full device reset the data in a blockram reverts to the original state.

  The optional **reserved** qualifier before the memory type, tells the linker not to locate any section in the memory by default. You can locate sections in such memories using an absolute address or range restriction (see subsection Locating a group in Section 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 **priority** field specifies a locate priority for a memory. The **speed** field has the same meaning but is considered deprecated. By default, a memory has its priority set to 1. The memories with the highest priority are considered first when trying to locate a rule. Subsequently, the next highest priority memories are added if the rule was not located successfully, and so on until the lowest priority that is available is reached or the rule is located. The lowest priority value is zero. Sections with an **ordered** and/or **contiguous** restriction are not affected by the locate priority. If such sections also have a **page** restriction, the locate priority is still used to select a page.
- If an **exec_priority** is specified for a memory, the regular priority (either specified or its default value) does not apply to locate rules with only executable sections. Instead, the supplied value applies for such rules. Additionally, the **exec_priority** value is used for any executable unrestricted sections, even if they appear in an unrestricted rule together with non-executable sections.
- The **map** field specifies how this memory maps onto an (internal) bus. The mapping can have a name. Mappings are described in Section 7.4.4, *Mappings*.
- The optional **write_unit** field specifies the minimum write unit (MWU). This is the minimum number of MAUs required in a write action. This is useful to initialize memories that can only be written in units of two or more MAUs. If **write_unit** is not defined the minimum write unit is 0.
- The optional **fill** field contains a bit pattern that the linker writes to all memory addresses that remain unoccupied during the locate process. The result of the expression, or list of expressions, is used as values to write to memory, each in MAU.

```
memory mem_name
{
   type = rom;
   mau = 8;
   write_unit = 4;
   fill = 0xaa;
   size = 64k;
   priority = 2;
   map map_name ( map_description );
}
```

With the **bus** keyword you define a bus in a derivative definition. Buses are described in Section 7.4.2, *Defining Internal Buses*.

# 7.6. Semantics of the Board Specification

## Keywords in the board specification

```
processor
   derivative
bus
   mau
   width
   map
memory
   type             reserved  rom  ram  nvram  blockram
   mau
   size
   speed
   priority
   exec_priority
   fill
   write_unit
   map

   map
      dest          bus  space
      dest_dbits
      dest_offset
      size
      src_dbits
      src_offset
      priority
      exec_priority
      reserved
```

## 7.6.1. Defining a Processor

If you have a target board with multiple processors that have the same derivative, you need to instantiate each individual processor in a processor definition. This information tells the linker which processor has which derivative and enables the linker to distinguish between the present processors.

If you use processors that all have a unique derivative, you may omit the processor definitions. In this case the linker assumes that for each derivative definition in the LSL file there is one processor. The linker uses the derivative name also for the processor.

With the keyword **processor** you define a processor. You can freely choose the processor name. The name is used to refer to it at other places in the LSL file:

```
processor proc_name
{
   processor definition
}
```

## 7.6.2. Instantiating Derivatives

With the keyword **derivative** you tell the linker that the given processor has a certain derivative. The derivative name refers to an existing derivative definition in the same LSL file.

For example, if you have two processors on your target board (called `myproc_1` and `myproc_2`) that have the same derivative (called `myderiv`), you must instantiate both processors as follows:

```
processor myproc_1
{
     derivative = myderiv;
}

processor myproc_2
{
     derivative = myderiv;
}
```

If the derivative definition has parameters you must specify the arguments that correspond with the parameters. For example `myderiv1` expects two parameters which are used in the derivative definition:

```
processor myproc
{
     derivative = myderiv1 (2,4);
}
```

## 7.6.3. Defining External Memory and Buses

It is common to define external memory (off-chip) and external buses at the global scope (outside any enclosing definition). Internal memory (on-chip memory) is usually defined in the scope of a derivative definition.

With the keyword **memory** you define physical memory that is present on the target board. The memory name is used to identify the memory and does not conflict with other identifiers. If you define memory parts in the LSL file, only the memory defined in these parts is used for placing sections.

If no external memory is defined in the LSL file and if the linker option to allocate memory on demand is set then the linker will assume that all virtual addresses are mapped on physical memory. You can override this behavior by specifying one or more memory definitions.

```
memory mem_name
{
   type = rom;
   mau = 8;
   write_unit = 4;
   fill = 0xaa;
   size = 64k;
   priority = 2;
   map map_name ( map_description );
}
```

For a description of the keywords, see Section 7.5.3, *Defining Internal Memory and Buses*.

With the keyword **bus** you define a bus (the combination of data and corresponding address bus). The bus name is used to identify a bus and does not conflict with other identifiers. Bus descriptions at the global scope (outside any definition) define external buses. These are buses that are present on the target board.

```
bus bus_name
{
     mau = 8;
     width = 8;
     map ( map_description );
}
```

For a description of the keywords, see Section 7.4.2, *Defining Internal Buses*.

You can connect off-chip memory to any derivative: you need to map the off-chip memory to a bus and map that bus on the internal bus of the derivative you want to connect it to.

# 7.7. Semantics of the Section Setup Definition

## Keywords in the section setup definition

```
section_setup
   stack
      min_size
      grows          low_to_high   high_to_low
      align
      fixed
      id
   heap
```

```
      min_size
      grows         low_to_high  high_to_low
      align
      fixed
      id
  reserved
```

### 7.7.1. Setting up a Section

With the keyword **section_setup** you can define stacks, heaps and/or reserved address ranges outside their address space definition.

```
section_setup ::my_space
{
   reserved address range
   stack definition
   heap definition
}
```

See the subsections Stacks and heaps and Reserved address ranges in Section 7.4.3, *Defining Address Spaces* for details on the keywords **stack**, **heap** and **reserved**.

## 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 p
   copy
   nocopy
   fill
   ordered
   contiguous
   clustered
   overlay
   allow_cross_references
   load_addr
      mem
   run_addr
      mem
   page
   page_size
   priority
select
stack
   size
```

```
heap
   size
reserved
   size
   attributes    r w x
   fill
   alloc_allowed absolute ranged
copytable
memcopy
   memory
   fill
section
   size
   blocksize
   attributes    r w x
   fill
   overflow

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
}
```

#### Locate direction

With the optional keyword **`direction`** you specify whether the linker starts locating sections from **`low_to_high`** (default) or from **`high_to_low`**. In the second case the linker starts locating sections at the highest addresses in the address space but preserves the order of sections when necessary (one processor and core in this example).

```
section_layout ::my_space ( direction = high_to_low )
{
   section statements
}
```

If you do not explicitly tell the linker how to locate a section, the linker decides on the basis of the section attributes in the object file and the information in the architecture definition and memory parts where to locate the section.

## 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 keyword **select** you can select one or more sections for the group. You can select a section by name or by attributes. If you select a section by name, you can use a wildcard pattern:

| | |
|---|---|
| * | matches with all section names |
| ? | matches with a single character in the section name |
| \ | takes the next character literally |
| [abc] | matches with a single 'a', 'b' or 'c' character |
| [a-z] | matches with any single character in the range 'a' to 'z' |

```
group ( ... )
{
   select "mysection";
   select "*";
}
```

The first **select** statement selects the section with the name "mysection". The second **select** statement selects all sections that were not selected yet.

A section is selected by the first select statement that matches, in the union of all section layouts for the address space. Global section layouts are processed in the order in which they appear in the LSL file. Internal core architecture section layouts always take precedence over global section layouts.

- The **`attributes`** field selects all sections that carry (or do not carry) the given attribute. With **+***attribute* you select sections that have the specified attribute set. With **-***attribute* you select sections that do not have the specified attribute set. You can specify one or more of the following attributes:
  - **r** readable sections
  - **w** writable sections
  - **x** executable sections
  - **i** initialized sections
  - **b** sections that should be cleared at program startup
  - **s** scratch sections (not cleared and not initialized)
  - **p** protected sections

To select all read-only sections:

```
group ( ... )
{
   select (attributes = +r-w);
}
```

Keep in mind that all section selections are restricted to the address space of the section layout in which this group definition occurs.

- With the **`ref_tree`** field you can select a group of related sections. The relation between sections is often expressed by means of references. By selecting just the 'root' of tree, the complete tree is selected. This is for example useful to locate a group of related sections in special memory (e.g. fast memory). The (referenced) sections must meet the following conditions in order to be selected:
  1. The sections are within the section layout's address space
  2. The sections match the specified attributes
  3. The sections have no absolute restriction (as is the case for all wildcard selections)

  For example, to select the code sections referenced from `foo1`:

  ```
  group refgrp (ordered, contiguous, run_addr=mem:ext_c)
  {
      select ref_tree "foo1" (attributes=+x);
  }
  ```

  If section `foo1` references `foo2` and `foo2` references `foo3`, then all these sections are selected by the selection shown above.

## Locating a group

```
group group_name ( group_specifications )
{
```

```
   section_statements
}
```

With the *group_specifications* you actually define how the linker must locate the group. You can roughly define three things: 1) assign properties to the sections in a group like alignment and read/write attributes, 2) define the mutual order in the address space for sections in the group and 3) restrict the possible addresses for the sections in a group.

The linker creates labels that allow you to refer to the begin and end address of a group from within the application software. Labels _**lc_gb**_*group_name* and _**lc_ge**_*group_name* mark the begin and end of the group respectively, where the begin is the lowest address used within this group and the end is the highest address used. Notice that a group not necessarily occupies all memory between begin and end address. The given label refers to where the section is located at run-time (versus load-time).

1. Assign properties to the sections in a group like alignment and read/write attributes.

   These properties are assigned to all sections in the group (and subgroups) and override the attributes of the input sections.

   - The **align** field tells the linker to align all sections in the group according to the align value. The alignment of a section is first determined by its own initial alignment and the defined alignment for the address space. Alignments are never decreased, if multiple alignments apply to a section, the largest one is used.

   - The **attributes** field tells the linker to assign one or more attributes to all sections in the group. This overrules the default attributes. By default the linker uses the attributes of the input sections. You can set the **r**, **w**, or **rw** attributes and you can switch between the **b** and **s** attributes.

   - The **copy** field tells the linker to locate a read-only section in RAM and generate a ROM copy and a copy action in the copy table. This property makes the sections in the group writable which causes the linker to generate ROM copies for the sections.

   - The effect of the **nocopy** field is the opposite of the **copy** field. It prevents the linker from generating ROM copies of the selected sections.

2. Define the mutual order of the sections in the group.

   By default, a group is *unrestricted* which means that the linker has total freedom to place the sections of the group in the address space.

   - The **ordered** keyword tells the linker to locate the sections in the same order in the address space as they appear in the group (but not necessarily adjacent).

     Suppose you have an ordered group that contains the sections 'A', 'B' and 'C'. By default the linker places the sections in the address space like 'A' - 'B' - 'C', where section 'A' gets the lowest possible address. With **direction=high_to_low** in the **section_layout** space properties, the linker places the sections in the address space like 'C' - 'B' - 'A', where section 'A' gets the highest possible address.

   - The **contiguous** keyword tells the linker to locate the sections in the group in a single address range. Within a contiguous group the input sections are located in arbitrary order, however the group occupies one contiguous range of memory. Due to alignment of sections there can be 'alignment gaps' between the sections.

When you define a group that is both **ordered** and **contiguous**, this is called a *sequential* group. In a sequential group the linker places sections in the same order in the address space as they appear in the group and it occupies a contiguous range of memory.

- The **clustered** keyword tells the linker to locate the sections in the group in a number of *contiguous* blocks. It tries to keep the number of these blocks to a minimum. If enough memory is available, the group will be located as if it was specified as **contiguous**. Otherwise, it gets split into two or more blocks.

  If a contiguous or clustered group contains *alignment gaps*, the linker can locate sections that are not part of the group in these gaps. To prevent this, you can use the **fill** keyword. If the group is located in RAM, the gaps are treated as reserved (scratch) space. If the group is located in ROM, the alignment gaps are filled with zeros by default. You can however change the fill pattern by specifying a bit pattern. The result of the expression, or list of expressions, is used as values to write to memory, each in MAU.

- The **overlay** keyword tells the linker to overlay the sections in the group. The linker places all sections in the address space using a contiguous range of addresses. (Thus an overlay group is automatically also a contiguous group.) To overlay the sections, all sections in the overlay group share the same run-time address.

  For each input section within the overlay the linker automatically defines two symbols. The symbol **_lc_cb_***section_name* is defined as the load-time start address of the section. The symbol **_lc_ce_***section_name* is defined as the load-time end address of the section. Assembly code may be used to copy the overlaid sections.

  If sections in the overlay group contain references between groups, the linker reports an error. The keyword **allow_cross_references** tells the linker to accept cross-references. Normally, it does not make sense to have references between sections that are overlaid.

```
group ovl (overlay)
{
    group a
    {
        select "my_ovl_p1";
        select "my_ovl_p2";
    }
    group b
    {
        select "my_ovl_q1";
    }
}
```

It may be possible that one of the sections in the overlay group already has been defined in another group where it received a load-time address. In this case the linker does not overrule this load-time address and excludes the section from the overlay group.

3. Restrict the possible addresses for the sections in a group.

   The load-time address specifies where the group's elements are loaded in memory at download time. The run-time address specifies where sections are located at run-time, that is when the program is executing. If you do not explicitly restrict the address in the LSL file, the linker assigns addresses to the sections based on the restrictions relative to other sections in the LSL file and section alignments. The program is responsible for copying overlay sections at appropriate moment from its load-time location to its run-time location (this is typically done by the startup code).

   - The **run_addr** keyword defines the run-time address. If the run-time location of a group is set explicitly, the given order between groups specify whether the run-time address propagates to the parent group or not. The location of the sections a group can be restricted either to a single absolute address, or to a number of address ranges (not including the end address). With an expression you can specify that the group should be located at the absolute address specified by the expression:

     ```
     group (run_addr = 0xa00f0000)
     ```

     If the group is ordered, the first section in the group is located at the specified absolute address.

     You can use the '**[***offset***]**' variant to locate the group at the given absolute offset in memory:

     ```
     group (run_addr = mem:A[0x1000])
     ```

     If the group is ordered, the first section in the group is located at the specified absolute offset in memory.

     A range can be an absolute space address range, written as **[** *expr* .. *expr* **]**, a complete memory device, written as **mem:***mem_name*, or a memory address range, **mem:**mem_name**[***expr* .. *expr* **]**

     ```
     group (run_addr = mem:my_dram)
     ```

     You can use the '**|**' to specify an address range of more than one physical memory device:

     ```
     group (run_addr = mem:A | mem:B)
     ```

     When used in top-level section layouts, a memory name refers to a board-level memory. You can select on-chip memory with **mem:***proc_name***:***mem_name*. If the memory has multiple parallel mappings towards the current address space, you can select a specific named mapping in the memory by appending */map_name* to the memory specifier. The linker then maps memory offsets only through that mapping, so the address(es) where the sections in the group are located are determined by that memory mapping.

     ```
     group (run_addr = mem:CPU1:A/cached)
     ```

   - The **load_addr** keyword changes the meaning of the section selection in the group: the linker selects the load-time ROM copy of the named section(s) instead of the regular sections. Just like **run_addr** you can specify an absolute address or an address range.

     ```
     group (contiguous, load_addr)
     {
       select "mydata";  // select ROM copy of mydata:
     ```

```
                    // "[mydata]"
}
```

The load-time and run-time addresses of a group cannot be set at the same time. If the load-time property is set for a group, the group (only) restricts the positioning at load-time of the group's sections. It is not possible to set the address of a group that has a not-unrestricted parent group.

The properties of the load-time and run-time start address are:

- At run-time, before using an element in an overlay group, the application copies the sections from their load location to their run-time location, but only if these two addresses are different. For non-overlay sections this happens at program start-up.
- The start addresses cannot be set to absolute values for unrestricted groups.
- For non-overlay groups that do not have an overlay parent, the load-time start address equals the run-time start address.
- For any group, if the run-time start address is not set, the linker selects an appropriate address.
- If an ordered group or sequential group has an absolute address and contains sections that have separate page restrictions (not defined in LSL), all those sections are located in a single page. In other cases, for example when an unrestricted group has an address range assigned to it, the paged sections may be located in different pages.

For overlays, the linker reserves memory at the run-time start address as large as the largest element in the overlay group.

- The **page** keyword tells the linker to place the group in one page. Instead of specifying a run-time address, you can specify a page and optional a page number. Page numbers start from zero. If you omit the page number, the linker chooses a page.

  The **page** keyword refers to pages in the address space as defined in the architecture definition.

- With the **page_size** keyword you can override the page alignment and size set on the address space. When you set the page size to zero, the linker removes simple (auto generated) page restrictions from the selected sections. See also the **page_size** keyword in Section 7.4.3, *Defining Address Spaces*.

- With the **priority** keyword you can change the order in which sections are located. This is useful when some sections are considered important for good performance of the application and a small amount of fast memory is available. The value is a number for which the default is 1, so higher priorities start at 2. Sections with a higher priority are located before sections with a lower priority, unless their relative locate priority is already determined by other restrictions like **run_addr** and **page**.

  ```
  group (priority=2)
  {
    select "importantcode1";
    select "importantcode2";
  }
  ```

## 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 keyword **`stack`** tells the linker to reserve memory for the stack. The name for the stack section refers to the stack as defined in the architecture definition. If no name was specified in the architecture definition, the default name is `stack`.

  With the keyword **`size`** you can specify the size for the stack. If the size is not specified, the linker uses the size given by the **`min_size`** argument as defined for the stack in the architecture definition. Normally the linker automatically tries to maximize the size, unless you specified the keyword **`fixed`**.

  ```
  group ( ... )
  {
     stack "mystack" ( size = 2k );
  }
  ```

  The linker creates two labels to mark the begin and end of the stack, `_`**`lc_ub`**`_`*`stack_name`* for the begin of the stack and `_`**`lc_ue`**`_`*`stack_name`* for the end of the stack. The linker allocates space for the stack when there is a reference to either of the labels.

  See also the **`stack`** keyword in Section 7.4.3, *Defining Address Spaces*.

### Heap

- The keyword **`heap`** tells the linker to reserve a dynamic memory range for the `malloc()` function. Each heap section has a name. With the keyword **`size`** you can change the size for the heap. If the **`size`** is not specified, the linker uses the size given by the **`min_size`** argument as defined for the heap in the architecture definition. Normally the linker automatically tries to maximize the size, unless you specified the keyword **`fixed`**.

  ```
  group ( ... )
  {
     heap "myheap" ( size = 2k );
  }
  ```

  The linker creates two labels to mark the begin and end of the heap, `_`**`lc_ub`**`_`*`heap_name`* for the begin of the heap and `_`**`lc_ue`**`_`*`heap_name`* for the end of the heap. The linker allocates space for the heap when a reference to either of the section labels exists in one of the input object files.

### Reserved section

- The keyword **`reserved`** tells the linker to create an area or section of a given size. The linker will not locate any other sections in the memory occupied by a reserved section, with some exceptions. Each reserved section has a name. With the keyword **`size`** you can specify a size for a given reserved area or section.

```
group ( ... )
{
   reserved "myreserved" ( size = 2k );
}
```

The optional **`fill`** field contains a bit pattern that the linker writes to all memory addresses that remain unoccupied during the locate process. The result of the expression, or list of expressions, is used as values to write to memory, each in MAU. The first MAU of the fill pattern is always the first MAU in the section.

By default, no sections can overlap with a reserved section. With **`alloc_allowed=absolute`** sections that are located at an absolute address due to an absolute group restriction can overlap a reserved section. The same applies for reserved sections with **`alloc_allowed=ranged`** set. Sections restricted to a fixed address range can also overlap a reserved section.

With the **`attributes`** field you can set the access type of the reserved section. The linker locates the reserved section in its space with the restrictions that follow from the used attributes, **r**, **w** or **x** or a valid combination of them. The allowed attributes are shown in the following table. A value between < and > in the table means this value is set automatically by the linker.

| **Properties set in LSL** | | **Resulting section properties** | | |
|---|---|---|---|---|
| **attributes** | **filled** | **access** | **memory** | **content** |
| x | yes | | <rom> | executable |
| r | yes | r | <rom> | data |
| r | no | r | <rom> | scratch |
| rx | yes | r | <rom> | executable |
| rw | yes | rw | <ram> | data |
| rw | no | rw | <ram> | scratch |
| rwx | yes | rw | <ram> | executable |

```
group ( ... )
{
   reserved "myreserved" ( size = 2k,
                attributes = rw, fill = 0xaa );
}
```

If you do not specify any attributes, the linker will reserve the given number of maus, no matter what type of memory lies beneath. If you do not specify a fill pattern, no section is generated.

The linker creates two labels to mark the begin and end of the section, `_`**`lc_ub`**`_`*`name`* for the begin of the section and `_`**`lc_ue`**`_`*`name`* for the end of the reserved section.

## Output sections

- The keyword **`section`** tells the linker to accumulate sections obtained from object files ("input sections") into an output section of a fixed size in the locate phase. You can select the input sections with **`select`** statements. You can use groups inside output sections, but you can only set the **`align`**, **`attributes`**, **`copy`** and **`load_addr`** properties and the **`load_addr`** property cannot have an address specified.

The **fill** field contains a bit pattern that the linker writes to all unused space in the output section. When all input sections have an image (code/data) you must specify a fill pattern. If you do not specify a fill pattern, all input sections must be scratch sections. The fill pattern is aligned at the start of the output section.

As with a reserved section you can use the **attributes** field to set the access type of the output section.

```
group ( ... )
{
   section "myoutput" ( size = 4k, attributes = rw,
                        fill = 0xaa )
   {
      select "myinput1";
      select "myinput2";
   }
}
```

The available room for input sections is determined by the **size**, **blocksize** and **overflow** fields. With the keyword **size** you specify the fixed size of the output section. Input sections are placed from output section start towards higher addresses (offsets). When the end of the output section is reached and one or more input sections are not yet placed, an error is emitted. If however, the **overflow** field is set to another output section, remaining sections are located as if they were selected for the overflow output section.

```
group ( ... )
{
  section "tsk1_data" (size=4k, attributes=rw, fill=0,
                       overflow = "overflow_data")
  {
          select ".data.tsk1.*"
  }
  section "tsk2_data" (size=4k, attributes=rw, fill=0,
                       overflow = "overflow_data")
  {
          select ".data.tsk2.*"
  }
  section "overflow_data" (size=4k, attributes=rx,
                           fill=0)
  {
  }
}
```

With the keyword **blocksize** , the size of the output section will adapt to the size of its content. For example:

```
group flash_area (run_addr = 0x10000)
{
   section "flash_code" (blocksize=4k, attributes=rx,
                         fill=0)
   {
     select "*.flash";
   }
}
```

If the content of the section is 1 mau, the size will be 4 kB, if the content is 11 kB, the section will be 12 kB, etc. If you use **`size`** in combination with **`blocksize`**, the **`size`** value is used as default (minimal) size for this section. If it is omitted, the default size will be of **`blocksize`**. It is not allowed to omit both **`size`** and **`blocksize`** from the section definition.

The linker creates two labels to mark the begin and end of the section, **`_lc_ub_`***`name`* for the begin of the section and **`_lc_ue_`***`name`* for the end of the output section.

When the **`copy`** property is set on an enclosing group, a ROM copy is created for the output section and the output section itself is made writable causing it to be located in RAM by default. For this to work, the output section and its input sections must be read-only and the output section must have a **`fill`** property.

### Copy table

- The keyword **`copytable`** tells the linker to select a section that is used as *copy table*. The content of the copy table is created by the linker. It contains the start address and length of all sections that should be initialized by the startup code.

  The linker creates two labels to mark the begin and end of the section, **`_lc_ub_table`** for the begin of the section and **`_lc_ue_table`** for the end of the copy table. The linker generates a copy table when a reference to either of the section labels exists in one of the input object files.

### Memory copy sections

- If a memory (usually RAM) needs to be initialized by a different core than the one(s) that will use it, a copy of the contents of the memory can be placed in a section using a **`memcopy`** statement in a **`section_layout`**. All data (including code) present in the specified memory is then placed in a new section with the provided name and appropriate attributes. Unused areas in the memory are filled in the section using the supplied fill pattern or with zeros if no fill pattern is specified. If the memory contains a memory copy section the result is undefined. The actual initialization of the memory at run-time needs to be done separately, this LSL feature only directs the linker to make the data located in the memory available for initialization. Note that a memory of type **`ram`** cannot hold initialized data, use type **`blockram`** instead.

## 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_cp" := "_lc_ub_table";
    // when the symbol _lc_cp occurs as an undefined reference
    // in an object file, the linker generates a copy table
}
```

### 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 ( exists( "mysection" ) )
      select "mysection";
   else
      reserved "myreserved" ( size=2k );
}
```