TSK3000 Embedded Tools Reference

TR0109 June 08, 2005

Table of Contents

Manual Purpose and Structure

Windows Users

The documentation explains and describes how to use the TASKING TSK3000 toolchain to program a TSK3000 processor.

You can use the tools either with the graphical Altium Designer or from the command line in a command prompt window.

Structure

The toolchain documentation consists of a user's manual (*Using the TSK3000 Embedded Tools*), which includes a Getting Started section, and a separate reference manual (this manual).

Start by reading the *Getting Started* in Chapter 1 of the user's manual.

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

Once you are familiar with these tools, you can use this reference manual to lookup specific options and details to make full use of the TASKING toolchain.

Short Table of Contents

Chapter 7: Linker Script Language

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

Chapter 8: MISRA C Rules

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

Conventions Used in this Manual

Notation for syntax

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

bold	Type this part of the syntax literally.
italics	Substitute the italic word by an instance. For example: *filename* Type the name of a file in place of the word *filename*.
{ }	Encloses a list from which you must choose an item.
[ ]	Encloses items that are optional. For example **c3000** [**-?**] Both **c3000** and **c3000 -?** are valid commands.
|	Separates items in a list. Read it as OR.
...	You can repeat the preceding item zero or more times.

Example

c3000 [*option*]... *filename*

You can read this line as follows: enter the command **c3000** with or without an option, follow this by zero or more options and specify a *filename*. The following input lines are all valid:

```
c3000 test.c
c3000 -g test.c
c3000 -g -s test.c
```

Not valid is:

```
c3000 -g
```

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

Icons

The following illustrations are used in this manual:

Note: notes give you extra information.

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

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

Command line: type your input on the command line.

Reference: follow this reference to find related topics.

Related Publications

C Standards

- ISO/IEC 9899:1999(E), Programming languages – C [ISO/IEC]
  More information on the standards can be found at `http://www.ansi.org`
- DSP–C, An Extension to ISO/IEC 9899:1999(E),
  Programming languages – C [TASKING, TK0071–14]

TASKING Tools

- Using the TSK3000 Embedded Tools
  [Altium, GU0111]
- TSK3000A 32–bit RISC Processor Core Reference
  [Altium, CR0121]

1 C Language

Summary

This chapter contains a complete overview of the C language extensions of the TASKING C compiler.

1.1 Introduction

The TASKING C compiler fully supports the ISO C standard but adds possibilities to program the special functions of the TSK3000.

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

- Data types
- Keywords
  - Memory type qualifiers
  - Function qualifiers
- Register usage
- Intrinsic functions
- Pragmas
- Predefined macros

1.2 Data Types

The TASKING C compiler for the TSK3000 architecture (**c3000**) supports the following fundamental data types:

Type	C Type	Size (bit)	Align (bit)	Limits
Boolean	`_Bool`	8	8	0 or 1

Character	`char`
	`signed char`
		8	8	-2^7 .. 2^7-1

	`unsigned char`	8	8	0 .. 2^8-1

Integral	`short`
	`signed short`
		16	16	-2^{15} .. $2^{15}-1$

	`unsigned short`	16	16	0 .. $2^{16}-1$
	`enum`	32	32	-2^{31} .. $2^{31}-1$

	`int`
	`signed int`
	`long`
	`signed long`
		32	32	-2^{31} .. $2^{31}-1$

	`unsigned int`
	`unsigned long`
		32	32	0 .. $2^{32}-1$

	`long long`
	`signed long long`
		64	32	-2^{63} .. $2^{63}-1$

	`unsigned long long`	64	32	0 .. $2^{64}-1$
Pointer	pointer to function or data	32	32	0 .. $2^{32}-1$
Floating-Point	`float`	32	32	$-3.402e^{38}$ .. $-1.175e^{-38}$ $1.175e^{-38}$ .. $3.402e^{38}$

	`double`
	`long double`
		64	32	$-1.798e^{308}$ .. $-2.225e^{-308}$
				$2.225e^{-308}$ .. $1.798e^{308}$

Table 1-1: Data Types for the TSK3000

1.3 Keywords

__asm()

With the `__asm` keyword you can use assembly instructions in the C source.

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

instruction_template	Assembly instructions that may contain parameters from the input list or output list in the form: **%***parm_nr*
%*parm_nr*[**.***regnum*]	Parameter number in the range 0 .. 9. With the optional **.***regnum* you can access an individual register from a register pair.
output_param_list	[[**"=**[**&**]*constraint_char***"(**C_expression**)**],...]
input_param_list	[[**"***constraint_char***"(**C_expression**)**],...]
&	Says that an output operand is written to before the inputs are read, so this output must not be the same register as any input.
constraint _char	Constraint character: the type of register to be used for the *C_expression*.
C_expression	Any C expression. For output parameters it must be an *lvalue*, that is, something that is legal to have on the left side of an assignment.
register_save_list	[[**"***register_name***"**],...]
register_name:q	Name of the register you want to reserve.

Constraint character	**Type**	**Operand**	**Remark**
R	general purpose register (64 bits)	$v0,$v1, $a0 .. $a3, $kt0, $kt1, $t0..$t9, $s0 .. $s8	Based on the specified register, a register pair is formed (64–bit). For example $v0:$v1.
r	general purpose register (32 bits)	$v0,$v1, $a0 .. $a3, $kt0, $kt1, $t0..$t9, $s0 .. $s8	
i	immediate value	*#value*	
l	label	*label*	
m	memory label	*variable*	stack or memory operand, a fixed address
H	multiply and devide register higher result	$hi	
L	multiply and devide register lower result	$lo	
number	other operand	same as %*number*	used when in– and output operands must be the same

Table 1–2: Available input/output operand constraints for the TSK3000

Section 2.4, *Using Assembly in the C Source*, in Chapter *C Language* of the user's manual.

__at()

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

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

Section 2.3.1, *Placing an Object at an Absolute Address*, in Chapter *C Language* of the user's manual.

Memory type qualifiers

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

You can specify the following memory types:

Qualifier	Description
`__no_sdata`	Direct addressable RAM
`__sdata`	Direct short addressable RAM (Small data, +/– 32kB offset from global pointer register $gp)

Table 1–3: Memory Type Qualifiers for the TSK3000

Section 2.3, *Memory Qualifiers* of the user's manual.

Function Qualifiers

inline
__noinline

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

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

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

For more information see section 2.7.3, *Inlining Functions: inline*, in Chapter *C Language* of the user's manual.

__interrupt()

With the function type qualifier `__interrupt()` you can declare a function as an interrupt service routine. The function type qualifier `__interrupt()` takes one or more vector numbers (0..31) as argument(s). All supplied vector numbers will be initialized to point to the interrupt function.

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

```
void __interrupt(vector_number[, vector_number]...)
isr( void )
{
...
}
```

1.4 Register Usage

Parameter passing

Function parameters are first passed via registers. If no more registers are available for a parameter, the compiler pushes parameters on the stack

Parameter Type	Parameter Number			
	1	2	3	4
_Bool	a0	a1	a2	a3
char	a0	a1	a2	a3
short	a0	a1	a2	a3
int / long	a0	a1	a2	a3
float	a0	a1	a2	a3
32–bit pointer	a0	a1	a2	a3
32–bit struct	a0	a1	a2	a3
long long	a0, a1	a1, a2	a2, a3	
double	a0, a1	a1, a2	a2, a3	
64–bit struct	a0, a1	a1, a2	a2, a3	

Table 1–4: Register usage for parameter passing

If a register corresponding to a parameter number is already in use the next register is used.

Function return types

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

Return Type	Register
_Bool	v0
char	v0
short	v0
int / long	v0
float	v0
32–bit pointer	v0
32–bit struct	v0
long long	v0, v1
double	v0, v1
64–bit struct	v0, v1

Table 1–5: Register usage for function return types

1.5 Intrinsic Functions

Intrinsic functions are predefined functions that are recognized by the compiler. The compiler then generates the most efficient assembly code for these functions.The compiler always inlines the corresponding assembly instructions in the assembly source rather than calling the function.

The TASKING TSK3000 C compiler recognizes the following intrinsic functions:

__alloc

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

Allocate memory. Same as library function malloc().

Returns: a pointer to space in external memory of `size` bytes length. NULL if there is not enough space left.

__break

```
volatile int __break(int val);
```

Generates the assembly `break` instruction. Val is an 20-bit value which will be encoded in the code field of the break instruction.

Returns: nothing.

__free

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

Deallocates the memory pointed to by *p*. *p* must point to memory earlier allocated by a call to `__alloc()`. Same as library function free().

Returns: nothing.

__nop

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

Generate NOP instructions.

Returns: nothing.

Example:

```
__nop();  /* generate NOP instruction */
```

__get_return_address

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

Used by the compiler for profiling when you compile with the **-p** (**--profile**) option.

Returns: return address of a function.

__mfc0

```
volatile int __mfc0(int spr);
```

Get the value from coprocessor 0 special function register *spr*.

Returns: the value of the *spr* register of coprocessor 0.

__mtc0

```
volatile void __mtc0(int val, int spr);
```

Put a value *val* into special purpose register *spr* of coprocessor 0.

Returns: nothing.

1.6 Pragmas

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

The syntax is:

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

or:

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

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

alias *symbol=defined_symbol*

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

See the **.EQU** directive directive in Section 3.2, *Assembler Directives and Controls*, in Chapter *Assembly Language*.

call {***near***|***far***}

Default, functions are called with 28-bit PC-region calls. This *near* call is directly coded into the instruction, resulting in higher execution speed and smaller code size.

The other call mode is a 32-bit absolute call. With *far* calls you can address the full range of memory. The address is first loaded into a register after which the call is executed.

Near calls are only possible if the destination address of the call is located within the same 256 MB region as the address of the call itself (hardware restriction). If you need to call a function (just) outside the 256 MB region from where it is called, you must use a far call.

See compiler option **-m (--call)** in section 4.1, *Compiler Options,* in Chapter *Tool Options*.

extern *symbol*

Force an external reference (**.extern** assembler directive), even when the symbol is not used in the module.

See the **.EXTERN** directive directive in Section 3.2, *Assembler Directives and Controls*, in Chapter *Assembly Language*.

inline
noinline
smartinline

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

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

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

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

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

smartinline

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

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

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

See for more information of the options **--inline-max-incr** and **--inline-max-size**, section 4.1, *Compiler Options* in Chapter *Tool Options*.

macro
nomacro

Turns macro expansion on or off. Default, macro expansion is enabled.

message *"message"* ***...***

Print the message string(s) on standard output.

optimize *flags*
endoptimize

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

See section 4.3, *Compiler Optimizations* in Chapter *Using the Compiler* in the user's manual. See compiler option **-O** in section 4.1, *Compiler Options,* in Chapter *Tool Options*.

runtime *[flag,...]*

Check for runtime errors. The pragma works the same as compiler option **-r (--runtime)**.

See compiler option **-r (--runtime)** in section 4.1, *Compiler Options* in Chapter *Tool Options* of the reference manual.

sdata *size*

With this pragma you tell the compiler to place all data objects smaller than the specified *size* (bytes) in sdata or sbss sections. You can still overrule this option with the keywords `__no_sdata` and `__sdata` for individiual data objects in your source.

See compiler option **--sdata** in section 4.1, *Compiler Options* in Chapter *Tool Options* of the reference manual.

section *name*

Use this pragma to rename sections. All sections are suffixed with the specified *name*.
For example:

```
#pragma section mysection
```

All sections are named `.text.mysection`, `.data.mysection` etc.

source
nosource

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

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

tradeoff *level*

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

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

warning *[number,...]*

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

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

weak *symbol*

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

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

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

See the **.WEAK** directive in Section 3.2, *Assembler Directives and Controls*, in Chapter *Assembly Language*.

1.7 Predefined Macros

In addition to the predefined macros required by the ISO C standard, such as `__DATA__` and `__FILE__`, the TASKING C compiler supports the predefined macros as defined in the table below. The macros are useful to create conditional C code.

Macro	Description
`__C3000__`	Expands to 1 for the TSK3000 toolchain, otherwise unrecognized as macro.
`__CPU__`	Expands to the CPU core name (option **–C***cpu*).
`__SINGLE_FP__`	Expands to 1 if you used option **–F** (Treat 'double' as 'float'), otherwise unrecognized as macro.
`__DOUBLE_FP__`	Expands to 1 if you did *not* use option **–F** (Treat 'double' as 'float'), otherwise unrecognized as macro.
`__TASKING__`	Identifies the compiler as a TASKING compiler. Expands to 1 if a TASKING compiler is used.
`__VERSION__`	Identifies the version number of the compiler. For example, if you use version 1.0r2 of the compiler, __VERSION__ expands to 1000 (dot and revision number are omitted, minor version number in 3 digits).
`__REVISION__`	Identifies the revision number of the compiler. For example, if you use version 1.0r2 of the compiler, __REVISION__ expands to 2.
`__BUILD__`	Identifies the build number of the compiler, composed of decimal digits for the build number, three digits for the major branch number and three digits for the minor branch number. For example, if you use build 1.22.1 of the compiler, __BUILD__ expands to 1022001. If there is no branch number, the branch digits expand to zero. For example, build 127 results in 127000000.

Table 1–6: Predefined macros

2 Libraries

Summary

This chapter lists all library functions that you can call in your C source.

2.1 Introduction

This chapter contains an overview of all library functions that you can call in your C source. This includes all functions of the standard C library (ISO C99) and some functions of the floating−point library.

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

The following table lists all available libraries:

Libraries	**Description**
c3000.lib c3000md.lib	C library (some functions also need the floating−point library) C library with support for hardware multiply/divide
c3000s.lib c3000mds.lib	Single precision C library (some functions also need the floating−point library)
fp3000.lib fp3000md.lib	Floating−point library (non trapping)
fp3000t.lib fp3000mdt.lib	Floating−point library (trapping)

pb3000.lib
pc3000.lib
pct3000.lib
pd3000.lib
pt3000.lib
p*3000md.lib
	Profiling libraries: pb = block/function counter
	pc = call graph
	pct = call graph and timing
	pd = dummy
	pt = function timing

Table 2−1: Overview of libraries

2.2 Library Functions

A number of wide-character functions are available as C source code, but have not been compiled with the C library. To use complete wide-character functionality, you must recompile the libraries with the macro WCHAR_SUPPORT_ENABLED and keep this macro also defined when compiling your own sources. (See compiler option **-D** (**--define**) in Chapter 4*, Tool options*.)

2.2.1 assert.h

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

2.2.2 complex.h

The TSK3000 does not support complex numbers.

2.2.3 ctype.h and wctype.h

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

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

Ctype.h	Wctype.h	Description
`toupper`	`towupper`	Returns c converted to an uppercase character if it is a lowercase character, otherwise c is returned.
`_tolower`	–	Converts c to a lowercase character, does not check if c really is an uppercase character. Implemented as macro. This macro function is not defined in ISO/IEC 9899.
`_toupper`	–	Converts c to an uppercase character, does not check if c really is a lowercase character. Implemented as macro. This macro function is not defined in ISO/IEC 9899.
`isascii`		Returns a non–zero value when c is in the range of 0 and 127. This function is not defined in ISO/IEC 9899.
`toascii`		Converts c to an ASCII value (strip highest bit). This function is not defined in ISO/IEC 9899.

2.2.4 errno.h

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

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

`EZERO`	`0`	No error
`EPERM`	`1`	Not owner
`ENOENT`	`2`	No such file or directory
`EINTR`	`3`	Interrupted system call
`EIO`	`4`	I/O error
`EBADF`	`5`	Bad file number
`EAGAIN`	`6`	No more processes
`ENOMEM`	`7`	Not enough core
`EACCES`	`8`	Permission denied
`EFAULT`	`9`	Bad address
`EEXIST`	`10`	File exists
`ENOTDIR`	`11`	Not a directory
`EISDIR`	`12`	Is a directory
`EINVAL`	`13`	Invalid argument
`ENFILE`	`14`	File table overflow
`EMFILE`	`15`	Too many open files
`ETXTBSY`	`16`	Text file busy
`ENOSPC`	`17`	No space left on device
`ESPIPE`	`18`	Illegal seek
`EROFS`	`19`	Read–only file system
`EPIPE`	`20`	Broken pipe
`ELOOP`	`21`	Too many levels of symbolic links
`ENAMETOOLONG`	`22`	File name too long

Floating–point errors

`EDOM`	23	Argument too large
`ERANGE`	24	Result too large

Errors returned by prinff/scanf

`ERR_FORMAT`	25	Illegal format string for printf/scanf
`ERR_NOFLOAT`	26	Floating–point not supported
`ERR_NOLONG`	27	Long not supported
`ERR_NOPOINT`	28	Pointers not supported

Error returned by file positioning routines

`ERR_POS`	29	Positioning failure

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

`EILSEQ`	30	Illegal byte sequence (including too few bytes)

2.2.5 fcntl.h

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

2.2.6 fenv.h

Contains mechanisms to control the floating–point environment.

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

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

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

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

Currently no rounding mode macros are implemented.

2.2.7 float.h

The header file `float.h` defines the characteristics of the real floating-point types `float`, `double` and `long double`.

`Float.h` used to contain prototypes for the functions `copysign(f)`, `isinf(f)`, `isfinite(f)`, `isnan(f)` and `scalb(f)`. These functions have accordingly to the ISO/IEC9899 standard been moved to the header file `math.h`. See also section 2.2.13, *Math.h and Tgmath.h*.

2.2.8 fss.h

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

Fss.h	**Description**
`_fss_break(`*`void`*`)`	Buffer and breakpoint functions for the debugger.
`_fss_init(`*`fd,is_close`*`)`	Opens file descriptors 0 (stdin), 1 (stdout) and 2 (stderr) and associates them with terminal window `FSS 0` of the debugger.

`_close(`*`fd`*`)`
`_lseek(`*`fd,offset,whence`*`)`
`_open(`*`fd,flags`*`)`
`_read(`*`fd,*buff,cnt`*`)`
`_unlink(`*`*name`*`)`
`_write(`*`fd,*buffer,cnt`*`)`
	See *Low Level File Access Functions* in section 2.2.20, *Stdio.h*.

2.2.9 inttypes.h and stdint.h

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

`intmax_t` **`imaxabs`**`(intmax_t j);`	Returns the absolute value of j
`imaxdiv_t` **`imaxdiv`**`(intmax_t numer, intmax_t denom);`	Computes numer/denom and numer % denom. The result is stored in the quot and rem components of the imaxdiv_t structure type.

`intmax_t` **`strtoimax`**`(const char * restrict nptr, char ** restrict endptr, int base);`	Convert string to maximum sized integer. (Compare `strtol`)
`uintmax_t` **`strtoumax`**`(const char * restrict nptr, char ** restrict endptr, int base);`	Convert string to maximum sized unsigned integer. (Compare `strtoul`)
`intmax_t` **`wcstoimax`**`(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);`	Convert wide string to maximum sized integer. (Compare `wctol`)
`uintmax_t` **`wcstoumax`**`(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);`	Convert wide string to maximem sized unsigned integer. (Compare `wctoul`)

2.2.10 iso646.h

The header file iso646.h adds tokens that can be used instead of regular operator tokens.

```
#define and     &&
#define and_eq  &=
#define bitand  &
#define bitor   |
#define compl   ~
#define not     !
#define not_eq  !=
#define or      ||
#define or_eq   |=
#define xor     ^
#define xor_eq  ^=
```

2.2.11 limits.h

Contains the sizes of integral types, defined as macros.

2.2.12 locale.h

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

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

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

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

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

```
struct lconv *localeconv( void )
```

Returns a pointer to type `stuct lconv` with values appropriate for the formatting of numeric quantities according to the rules of the current locale. The `struct lconv` in this header file is conforming the ISO standard.

2.2.13 math.h and tgmath.h

The header file `math.h` contains the prototypes for many mathematical functions. Before C99, all functions were computed using the double type (the float was automatically converted to double, prior to calculation). In this C99 version, parallel sets of functions are defined for double, float and long double. They are respectively named *function*, *function*`f`, *function*`l`. All `long` type functions, though declared in `math.h`, are implemented as the `double` type variant which nearly always meets the requirement in embedded applications.

The header file `tgmath.h` contains parallel type generic math macros whose expansion depends on the used type. `tgmath.h` includes `math.h` and the effect of expansion is that the correct `math.h` functions are called. The type generic macro, if available, is listed in the second column of the tables below.

Trigonometric functions

Math.h			**Tgmath.h**	**Description**
`sin`	`sinf`	`sinl`	`sin`	Returns the sine of `x`.
`cos`	`cosf`	`cosl`	`cos`	Returns the cosine of `x`.
`tan`	`tanf`	`tanl`	`tan`	Returns the tangent of `x`.
`asin`	`asinf`	`asinl`	`asin`	Returns the arc sine $\sin^{-1}(x)$ of `x`.
`acos`	`acosf`	`acosl`	`acos`	Returns the arc cosine $\cos^{-1}(x)$ of `x`.
`atan`	`atanf`	`atanl`	`atan`	Returns the arc tangent $\tan^{-1}(x)$ of `x`.
`atan2`	`atan2f`	`atan2l`	`atan2`	Returns the result of: $\tan^{-1}(y/x)$.
`sinh`	`sinhf`	`sinhl`	`sinh`	Returns the hyperbolic sine of `x`.
`cosh`	`coshf`	`coshl`	`cosh`	Returns the hyperbolic cosine of `x`.
`tanh`	`tanhf`	`tanhl`	`tanh`	Returns the hyperbolic tangent of `x`.
`asinh`	`asinhf`	`asinhl`	`asinh`	Returns the arc hyperbolic sinus of `x`.
`acosh`	`acoshf`	`acoshl`	`acosh`	Returns the non–negative arc hyperbolic cosinus of `x`.
`atanh`	`atanhf`	`atanhl`	`atanh`	Returns the arc hyperbolic tangent of `x`.

Exponential and logarithmic functions

All of these functions are new in C99, except for `exp`, `log` and `log10`.

Math.h			Tgmath.h	Description
`exp`	`expf`	`expl`	`exp`	Returns the result of the exponential function e^x.
`exp2`	`exp2f`	`exp2l`	`exp2`	Returns the result of the exponential function 2^x. (*Not implemented*)
`expm1`	`expm1f`	`expm1l`	`expm1`	Returns the result of the exponential function e^x-1 (*Not implemented*)
`log`	`logf`	`logl`	`log`	Returns the natural logarithm `ln(x), x>0`.
`log10`	`log10f`	`log10l`	`log10`	Returns the base-10 logarithm of `x, x>0`.
`log1p`	`log1pf`	`log1pl`	`log1p`	Returns the base-e logarithm of `(1+x). x <> -1`. (*Not implemented*)
`log2`	`log2f`	`log2l`	`log2`	Returns the base-2 logarithm of `x. x>0`. (*Not implemented*)
`ilogb`	`ilogbf`	`ilogbl`	`ilogb`	Returns the signed exponent of x as an integer. `x>0`. (*Not implemented*)
`logb`	`logbf`	`logbl`	`logb`	Returns the exponent of `x` as a signed integer in value in floating-point notation. `x > 0`. (*Not implemented*)

Rounding functions

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

Remainder after devision

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

frexp, ldexp, modf, scalbn, scalbln

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

Power and absolute-value functions

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

Manipulation functions: copysign, nan, nextafter, nexttoward

Math.h			Tgmath.h	Description
`copysign`		`copysignl` `copysignf`	`copysign`	Returns the value of `x` with the sign of `y`.
`nan`	`nanl`	`nanf`	–	Returns a quiet NaN, if available, with content indcated through `tagp`. (*Not implemented*)
`nextafter`		`nextafterl` `nextafterf`	`nextafter`	Returns the next representable value in the specified format after `x` in the direction of `y`. Returns `y` is `x=y`. (*Not implemented*)
`nexttoward`		`nexttowardl` `nexttowardf`	`nexttoward`	Same as `nextafter`, except that the second argument in all three variants is of type long double. Returns y if `x=y`. (*Not implemented*)

Positive difference, maximum, minimum

Math.h			Tgmath.h	Description
fdim	fdiml	fdimf	fdim	Returns the positive difference between: |x−y|. (*Not implemented*)
fmax	fmaxl	fmaxf	fmax	Returns the maximum value of their arguments. (*Not implemented*)
fmin	fminl	fminf	fmin	Returns the minimum value of their arguments. (*Not implemented*)

Error and gamma (Not implemented)

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

Comparison macros

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

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

Classification macros

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

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

2.2.14 setjmp.h

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

`int setjmp(jmp_buf env)`	Records its caller's environment in `env` and returns 0.
`void longjmp(jmp_buf env, int status)`	Restores the environment previously saved with a call to `setjmp()`.

2.2.15 signal.h

Signals are possible asynchronous events that may require special processing. Each signal is named by a number. The following signals are defined:

`SIGINT`	1	Receipt of an interactive attention signal
`SIGILL`	2	Detection of an invalid function message
`SIGFPE`	3	An errouneous arithmetic operation (for example, zero devide, `overflow)`
`SIGSEGV`	4	An invalid access to storage
`SIGTERM`	5	A termination request sent to the program
`SIGABRT`	6	Abnormal terminiation, such as is initiated by the `abort` function.

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

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

The next function determines how subsequent signals will be handled:

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

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

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

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

2.2.16 stdarg.h

The facilities in this header file gives you a portable way to access variable arguments lists, such as needed for as `fprintf` and `vfprintf`. This header file contains the following macros:

`va_arg(ap,type)`	Returns the value of the next argument in the variable argument list. It's return type has the type of the given argument `type`. A next call to this macro will return the value of the next argument.
`va_end(va_list ap)`	This macro must be called after the arguments have been processed. It should be called before the function using the macro 'va_start' is terminated (ANSI specification).
`va_start( va_list ap, lastarg );`	This macro initializes `ap`. After this call, each call to va_arg() will return the value of the next argument. In our implementation, `va_list` cannot contain any bit type variables. Also the given argument `lastarg` must be the last non–bit type argument in the list.

2.2.17 stdbool.h

This header file contains the following macro definitions. These names for boolean type and values are consisten with C++. You are allowed to `#undefine` or redefine the macros below.

```
#define bool                            _Bool
#define true                            1
#define false                           0
#define __bool_true_false_are_defined   1
```

2.2.18 stddef.h

This header file defines the types for common use:

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

Besides these types, the following macros are defined:

`NULL`	expands to the null pointer constant
`offsetof(_type,_member)`	expands to an integer constant expression with type `size_t` that is the offset in bytes of `_member` within structure type `_type`.

2.2.19 stdint.h

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

2.2.20 stdio.h and wchar.h

Types

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

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

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

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

Macros

Stdio.h	Description
`BUFSIZ 512`	Size of the buffer used by the `setbuf`/`setvbuf` function: 512
`EOF -1`	End of file indicator.
`WEOF UINTMAX`	End of file indicator. NOTE: WEOF need not to be a negative number as long as its value does not correspond to a member of the wide character set. (Defined in `wchar.h`).
`FOPEN_MAX`	Number of files that can be opened simultaneously: 4 NOTE: According to ISO/IEC 9899 this value must be at least 8.
`FILENAME_MAX 100`	Maximum length of a filename: 100

`_IOFBF`
`_IOLBF`
`_IONBF`
	Expand to an integer expression, suitable for use as argument to the `setvbuf` function.

`L_tmpnam`	Size of the string used to hold temporary file names: 8 (tmp*xxxxx*)
`TMP_MAX 0x8000`	Maximum number of unique temporary filenames that can be generated: 0x8000

`stderr`
`stdin`
`stdout`
	Expressions of type ”pointer to FILE” that point to the FILE objects associated with standard error, input and output streams.

Low level file access functions

Stdio.h	Description
`_close(`*`fd`*`)`	Used by the functions close and fclose. (*FSS implementation)*
`_lseek(`*`fd`*`,`*`offset`*`,`*`whence`*`)`	Used by all file positioning functions: fgetpos, fseek, fsetpos, ftell, rewind. (*FSS implementation)*
`_open(`*`fd`*`,`*`flags`*`)`	Used by the functions `fopen` and `freopen`. (*FSS implementation)*
`_read(`*`fd`*`,`*`*buff`*`,`*`cnt`*`)`	Reads a sequence of characters from a file. (*FSS implementation)*
`_unlink(`*`*name`*`)`	Used by the function remove. (*FSS implementation)*
`_write(`*`fd`*`,`*`*buffer`*`,`*`cnt`*`)`	Writes a sequence of characters to a file. (*FSS implementation)*

File access

Stdio.h	Description
`fopen(`*`name`*`,`*`mode`*`)`	Opens a file for a given mode. Available modes are: ”r” read; open text file for reading ”w” write; create text file for writing; if the file already exists its contents is discarded ”a” append; open existing text file or create new text file for writing at end of file ”r+” open text file for update; reading and writing ”w+” create text file for update; previous contents if any is discarded ”a+” append; open or create text file for update, writes at end of file
`fclose(`*`name`*`)`	Flushes the data stream and closes the specified file that was previously opened with fopen.
`fflush(`*`name`*`)`	If stream is an output stream, any buffered but unwritten date is written. Else, the effect is undefined.
`freopen(`*`name`*`,`*`mode`*`,`*`stream`*`)`	Similar to fopen, but rather then generating a new value of type FILE *****, the existing value is associated with a new stream.
`setbuf(`*`stream`*`,`*`buffer`*`)`	If buffer is NULL, buffering is turned off for the stream. Otherwise, setbuf is equivalent to: `(void) setvbuf(`*`stream`*`,`*`buf`*`,_IOFBF, BUFSIZ).`

`setvbuf(`*`stream`*`,`*`buffer`*`,`
`mode``,`*`size`*`)`
	Controls buffering for the *stream*; this function must be called before reading or writing. *Mode* can have the following values:
	`_IOFBF` causes full buffering
	`_IOLBF` causes line buffering of text files
	`_IONBF` causes no buffering
	If buffer is not NULL, it will be used as a buffer; otherwise a buffer will be allocated. *size* determines the buffer size.

Character input/output

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

- Flags (in any order):

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

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

Flags, length specifier, period, precision and length modifier are optional, the conversion character is not. The conversion character must be one of the following, if a character following '%' is not in the list, the behavior is undefined.

Character	Printed as
d, i	int, signed decimal
o	int, unsigned octal
x, X	int, unsigned hexadecimal in lowercase or uppercase respectively
u	int, unsigned decimal
c	int, single character (converted to unsigned char)
s	char *, the characters from the string are printed until a NULL character is found. When the given precision is met before, printing will also stop
f	double
e, E	double
g, G	double
n	int *, the number of characters written so far is written into the argument. This should be a pointer to an integer in default memory. No value is printed.
p	pointer (hexadecimal 24–bit value)
%	No argument is converted, a '%' is printed.

Table 2–2: Printf conversion characters

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

The format string can contain :

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

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

- A '*', meaning that no assignment is done for this field.
- A number specifying the maximum field width.
- The conversion characters `d`, `i`, `n`, `o`, `u` and `x` may be preceded by 'h' if the argument is a pointer to `short` rather than `int`, or by 'l' (letter ell) if the argument is a pointer to `long`, or by 'll' for a pointer to `long long`. The conversion characters `e`, `f`, and `g` may be preceded by 'l' if the argument is a pointer to `double` rather than `float`, and by 'L' for a pointer to a `long double`.
- A conversion specifier. '*', maximum field width and length modifier are optional, the conversion character is not. The conversion character must be one of the following, if a character following '%' is not in the list, the behavior is undefined.

Length specifier and length modifier are optional, the conversion character is not. The conversion character must be one of the following, if a character following '%' is not in the list, the behavior is undefined.

Character	Scanned as
d	int, signed decimal.
i	int, the integer may be given octal (i.e. a leading 0 is entered) or hexadecimal (leading ”0x” or ”0X”), or just decimal.
o	int, unsigned octal.
u	int, unsigned decimal.
x	int, unsigned hexadecimal in lowercase or uppercase.
c	single character (converted to unsigned char).
s	char *, a string of non white space characters. The argument should point to an array of characters, large enough to hold the string and a terminating NULL character.
f	float
e, E	float
g, G	float
n	int *, the number of characters written so far is written into the argument. No scanning is done.
p	pointer; hexadecimal 24–bit value which must be entered without 0x– prefix.
[...]	Matches a string of input characters from the set between the brackets. A NULL character is added to terminate the string. Specifying []...] includes the ’]’ character in the set of scanning characters.
[^...]	Matches a string of input characters not in the set between the brackets. A NULL character is added to terminate the string. Specifying [^]...] includes the ’]’ character in the set.
%	Literal ’%’, no assignment is done.

Table 2–3: Scanf conversion characters

Stdio.h	Wchar.h	Description
`fgetc(`*`stream`*`)`	`fgetwc(`*`stream`*`)`	Reads one character from *stream*. Returns the read character, or EOF/WEOF on error. (*FSS implementation*)
`getc(`*`stream`*`)`	`getwc(`*`stream`*`)`	Same as `fgetc`/`fgetwc` except that is implemented as a macro. (*FSS implementation*) NOTE: Currently #defined as `getchar()`/`getwchar()` because FILE I/O is not supported. Returns the read character, or EOF/WEOF on error.
`getchar(stdin)`	`getwchar(stdin)`	Reads one character from the `stdin` stream. Returns the character read or EOF/WEOF on error. Implemented as macro. (*FSS implementation*)

Stdio.h	Wchar.h	Description
`fgets(*s,n, stream)`	`fgetws(*s,n, stream)`	Reads at most the next *n*−1 characters from the *stream* into array *s* until a newline is found. Returns s or NULL or EOF/WEOF on error. (*FSS implementation*)
`gets(*s,n,stdin)`	–	Reads at most the next *n*−1 characters from the `stdin` stream into array *s*. A newline is ignored. Returns *s* or NULL or EOF/WEOF on error. (*FSS implementation*)
`ungetc(c,stream)`	`ungetwc(c,stream)`	Pushes character *c* back onto the input *stream*. Returns EOF/WEOF on error.
`fscanf(stream, format,...)`	`fwscanf(stream, format,...)`	Performs a formatted read from the given *stream*. Returns the number of items converted succesfully. (*FSS implementation*)
`scanf(format,...)`	`wscanf(format,...)`	Performs a formatted read from the `stdin` stream. Returns the number of items converted succesfully. (*FSS implementation*)
`sscanf(*s, format,...)`	`swscanf(*s, format,...)`	Performs a formatted read from the string *s*. Returns the number of items converted succesfully.
`vfscanf(stream, format,arg)`	`vfwscanf(stream, format,arg)`	Same as `fscanf/fwscanf`, but extra arguments are given as variable argument list *arg*. (See section 2.2.16, *stdarg.h*)
`vscanf(format,arg)`	`vwscanf(format,arg)`	Same as `scanf/wscanf`, but extra arguments are given as variable argument list *arg*. (See section 2.2.16, *stdarg.h*)
`vsscanf(s,format, arg)`	`vswscanf(s,format, arg)`	Same as `scanf/wscanf`, but extra arguments are given as variable argument list *arg*. (See section 2.2.16, *stdarg.h*)
`fputc(c,stream)`	`fputwc(c,stream)`	Put character *c* onto the given *stream*. Returns EOF/WEOF on error. (*FSS implementation*)
`putc(c,stream)`	`putwc(c,stream)`	Same as `fpuc/fputwc` except that is implemented as a macro. (*FSS implementation*)
`putchar(c,stdout)`	`putwchar(c,stdout)`	Put character *c* onto the `stdout` stream. Returns EOF/WEOF on error. Implemented as macro. (*FSS implementation*)
`fputs(*s,stream)`	`fputws(*s,stream)`	Writes string *s* to the given *stream*. Returns EOF/WEOF on error.
`puts(*s)`	–	Writes string *s* to the `stdout` stream. Returns EOF/WEOF on error. (*FSS implementation*)
`fprintf(stream, format,...)`	`fwprintf(stream, format,...)`	Performs a formatted write to the given *stream*. Returns EOF/WEOF on error. (*FSS implementation*)

Stdio.h	Wchar.h	Description
`printf(`*format*`,...)`	`wprintf(`*format*`,...)`	Performs a formatted write to the stream `stdout`. Returns EOF/WEOF on error. (*FSS implementation*)
`sprintf(*`*s*`,` *format*`,...)`	–	Performs a formatted write to string *s*. Returns EOF/WEOF on error.
`snprintf(*`*s*`,`*n* *format*`,...)`	`swprintf(*`*s*`,`*n* *format*`,...)`	Same as `sprintf`, but n specifies the maximum number of characters (including the terminating null character) to be written.
`vfprintf(`*stream*`,` *format*`,`*arg*`)`	`vfwprintf(`*stream*`,` *format*`,`*arg*`)`	Same as `fprintf`/`fwprintf`, but extra arguments are given as variable argument list *arg*. (See section 2.2.16, *stdarg.h*) (*FSS implementation*)
`vprintf(`*format*`,` *arg*`)`	`vwprintf(`*format*`,` *arg*`)`	Same as `printf`/`wprintf`, but extra arguments are given as variable argument list *arg*. (See section 2.2.16, *stdarg.h*) (*FSS implementation*)
`vsprintf(*`*s*`,` *format*`,`*arg*`)`	`vswprintf(*`*s*`,` *format*`,`*arg*`)`	Same as `sprintf`/`swprintf`, but extra arguments are given as variable argument list *arg*. (See section 2.2.16, *stdarg.h*) (*FSS implementation*)

Direct input/output

Stdio.h	Description
`fread(ptr,size,nobj,stream)`	Reads *nobj* members of *size* bytes from the given *stream* into the array pointed to by *ptr*. Returns the number of elements succesfully read. (*FSS implementation*)
`fwrite((ptr,size,nobj,stream)`	Writes *nobj* members of *size* bytes from to the array pointed to by *ptr* to the given *stream*. Returns the number of elements succesfully written. (*FSS implementation*)

Random access

Stdio.h	Description
`fseek(`*stream*`,`*offset*`,` *origin*`)`	Sets the position indicator for *stream*. (*FSS implementation*)

When repositioning a binary file, the new position *origin* is given by the following macros:

`SEEK_SET 0`	*offset* characters from the beginning of the file
`SEEK_CUR 1`	*offset* characters from the current position in the file
`SEEK_END 2`	*offset* characters from the end of the file

`ftell(stream)`	Returns the current file position for *stream*, or −1L on error. (*FSS implementation*)

`rewind(stream)`	Sets the file position indicator for the *stream* to the beginning of the file. This function is equivalent to:
	`(void) fseek( stream, 0L, SEEK_SET );`
	`clearerr( stream );`
	(*FSS implementation*)

`fgetpos(stream,pos)`	Stores the current value of the file position indicator for *stream* in the object pointed to by *pos*. (*FSS implementation*)
`fsetpos(stream,pos)`	Positions `stream` at the position recorded by `fgetpos` in **pos*. (*FSS implementation*)

Operations on files

Stdio.h	**Description**
`remove(file)`	Removes the named file, so that a subsequent attempt to open it fails. Returns a non−zero value if not succesful.
`rename(old,new)`	Changes the name of the file from old name to new name. Returns a non−zero value if not succesful.
`tmpfile()`	Creates a temporary file of the mode ”`wb+`” that will be automatically removed when closed or when the program terminates normally. Returns a `file` pointer.
`tmpnam(buffer)`	Creates new file names that do not conflict with other file names currently in use. The new file name is stored in a *buffer* which must have room for L_tmpnam characters. Returns a pointer to the temporary name. The file names are created in the current directory and all start with ”tmp”. At most TMP_MAX unique file names can be generated.

Error handling

Stdio.h	**Description**
`clearerr(stream)`	Clears the end of file and error indicators for stream.
`ferror(stream)`	Returns a non−zero value if the error indicator for stream is set.
`feof(stream)`	Returns a non−zero value if the end of file indicator for stream is set.
`perror(*s)`	Prints *s* and the error message belonging to the integer `errno`. (See section 2.2.4, *errno.h*)

2.2.21 stdlib.h and wchar.h

The header file `stdlib.h` contains general utility functions which fall into the following categories (Some have parallel wide−character, declared in `wchar.h`)

- Numeric conversions
- Random number generation
- Memory management
- Envirnoment communication
- Searching and sorting
- Integer arithmetic
- Multibyte/wide character and string conversions.

Macros

`RAND_MAX 32767`	Highest number that can be returned by the `rand`/`srand` function.

`EXIT_SUCCES 0`
`EXIT_FAILURE 1`
	Predefined exit codes that can be used in the `exit` function.

`MB_CUR_MAX 1`	Maximum number of bytes in a multibyte character for the extended character set specified by the current locale (category LC_CTYPE, see section 2.2.12, *locale.h*).

Numeric conversions

Next convert the intial portion of a string **s* to a `double`, `int`, `long int` and `long long int` value respectively.

```
double      atof(*s)
int         atoi(*s)
long        atol(*s)
long long   atoll(*s)
```

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

Stdlib.h

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

Wchar.h

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

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

Stdlib.h

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

Wchar.h

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

Random number generation

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

Memory management

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

Environment communication

`abort()`	Causes abnormal program termination. If the signal SIGABRTis caught, the signal handler may take over control. (See section 2.2.15, *signal.h*).
`atexit(*`*`func`*`)`	*Func* points to a function that is called (without arguments) when the program normally terminates.
`exit(`*`status`*`)`	Causes normal program termination. Acts as if `main()` returns with status as the return value. Status can also be specified with the predefined macros EXIT_SUCCES or EXIT_FAILURE.
`_Exit(`*`status`*`)`	Same as `exit`, but no registered by the `atexit` function or signal handlers registerd by the `signal` function are called.
`getenv(*s)`	Searches an environment list for a string *s*. Returns a pointer to the contents of *s*. NOTE: this function is not implemented because there is no OS.
`system(*s)`	Passes the string *s* to the environment for execution. NOTE: this function is not implemented because there is no OS.

Searching and sorting

`bsearch(*key,*base, n,size,*cmp)`	This function searches in an array of *n* members, for the object pointed to by *key*. The initial base of the array is given by *base*. The size of each member is specified by *size*. The given array must be sorted in ascending order, according to the results of the function pointed to by *cmp*. Returns a pointer to the matching member in the array, or NULL when not found.
`qsort(*base,n, size,*cmp)`	This function sorts an array of *n* members using the quick sort algorithm. The initial base of the array is given by *base*. The size of each member is specified by *size*. The array is sorted in ascending order, according to the results of the function pointed to by *cmp*.

Integer arithmetic

`int`	`abs(j)`	Compute the absolute value of an `int`, `long int`, and `long long int` *j* resepectively.
`long`	`labs(j)`	
`long long`	`llabs(j)`	
`div_t`	`div(x,y)`	Compute *x*/*y* and *x*%*y* in a single operation. *X* and *y* have respectively type `int`, `long int` and `long long int`. The result is stored in the members `quot` and `rem` of `struct div_t`, `ldiv_t` and `lldiv_t` which have the same types.
`ldiv_t`	`ldiv(x,y)`	
`lldiv_t`	`lldiv(x,y)`	

Multibyte/wide character and string conversions

`mblen(*s,n)`	Determines the number of bytes in the multi−byte character pointed to by *s*. At most *n* characters will be examined. (See also `mbrlen` in section 2.2.25, *wchar.h*)
`mbtowc(*pwc,*s,n)`	Converts the multi−byte character in *s* to a wide−character code and stores it in pwc. At most *n* characters will be examined.
`wctomb(*s,wc)`	Converts the wide−character *wc* into a multi−byte representation and stores it in the string pointed to by *s*. At most MB_CUR_MAX characters are stored.
`mbstowcs(*pwcs,*s,n)`	Converts a sequence of multi−byte characters in the string pointed to by *s* into a sequence of wide characters and stores at most *n* wide characters into the array pointed to by *pwcs*. (See also `mbsrtowcs` in section 2.2.25, *wchar.h*)
`wcstombs(*s,*pwcs,n)`	Converts a sequence of wide characters in the array pointed to by *pwcs* into multi−byte characters and stores at most *n* multi−byte characters into the string pointed to by *s*. (See also `wcsrtowmb` in section 2.2.25, *wchar.h*)

2.2.22 string.h and wchar.h

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

Copying and concatenation functions

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

Comparison functions

Stdio.h	Wchar.h	Description
`memcmp(*s1,*s2,n)`	`wmemcmp(*s1,*s2,n)`	Compares the first *n* characters of **s1* to the first *n* characters of **s2*. Returns < 0 if **s1* < **s2*, 0 if **s1* = = **s2*, or > 0 if **s1* > **s2*.
`strcmp(*s1,*s2)`	`wcscmp(*s1,*s2)`	Compares string **s1* to string **s2*. Returns < 0 if **s1* < **s2*, 0 if **s1* = = **s2*, or > 0 if **s1* > **s2*.
`strncmp(*s1,*s2,n)`	`wcsncmp(*s1,*s2,n)`	Compares the first *n* characters of **s1* to the first *n* characters of **s2*. Returns < 0 if **s1* < **s2*, 0 if **s1* = = **s2*, or > 0 if **s1* > **s2*.
`strcoll(*s1,*s2)`	`wcscoll(*s1,*s2)`	Performs a local-specific comparison between string **s1* and string **s2* according to the LC_COLLATE category of the current locale. Returns < 0 if **s1* < **s2*, 0 if **s1* = = **s2*, or > 0 if **s1* > **s2*. (See section 2.2.12, *locale.h*)

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

Search functions

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

Miscellaneous functions

Stdio.h	Wchar.h	Description
`memset(*s,c,n)`	`wmemset(*s,c,n)`	Fills the first *n* bytes of **s* with character *c* and returns **s*.
`strerror(errno)`	–	Typically, the values for errno come from `int errno`. This function returns a pointer to the associated error message. (See also section 2.2.4, *errno.h*)
`strlen(*s)`	`wcslen(*s)`	Returns the length of string **s*.

2.2.23 time.h and wchar.h

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

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

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

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

Time manipulation

`clock`	Returns the application's best approximation to the processor time used by the program since it was started. This low-level routine is not implemented because it strongly depends on the hardware. To determine the time in seconds, the result of clock should be divided by the value defined as `CLOCKS_PER_SEC   12000000`
`difftime(`*`t1`*`,`*`t0`*`)`	Returns the difference *t1–t0* in seconds.
`mktime(tm *`*`tp`*`)`	Converts the broken-down time in the structure pointed to by *tp*, to a value of type `time_t`. The return value has the same encoding as the return value of the `time` function.
`time(*`*`timer`*`)`	Returns the current calendar time. This value is also assigned to *`timer`.

Time conversion

`asctime(tm *`*`tp`*`)`	Converts the broken-down time in the structure pointed to by *tp* into a string in the form `Mon Jan 21 16:15:14 2004\n\0`. Returns a pointer to this string.
`ctime(*`*`timer`*`)`	Converts the calender time pointed to by *timer* to local time in the form of a string. This is equivalent to: `asctime(localtime(timer))`
`gmtime(*`*`timer`*`)`	Converts the calender time pointed to by *timer* to the broken-down time, expressed as UTC. Returns a pointer to the broken-down time.
`localtime(*`*`timer`*`)`	Converts the calendar time pointed to by *timer* to the broken-down time, expressed as local time. Returns a pointer to the broken-down time.

Formatted time

The next function has a parallel function defined in `wchar.h`:

Stdio.h	**Wchar.h**
`strftime(*s,smax,*fmt,tm *tp)`	`wstrftime(*s,smax,*fmt,tm *tp)`

Formats date and time information from `struct tm` **tp* into **s* according to the specified format **fmt*. No more than *smax* characters are placed into **s*. The formatting of `strftime` is locale-specific using the `LC_TIME` category (see section 2.2.12, *locale.h*). You can use the next conversion specifiers:

%a	abbreviated weekday name
%A	full weekday name
%b	abbreviated month name
%B	full month name
%c	local date and time representation
%d	day of the month (01–31)
%H	hour, 24-hour clock (00–23)
%I	hour, 12-hour clock (01–12)
%j	day of the year (001–366)
%m	month (01–12)
%M	minute (00–59)
%p	local equivalent of AM or PM
%S	second (00–59)
%U	week number of the year, Sunday as first day of the week (00–53)
%w	weekday (0–6, Sunday is 0)
%W	week number of the year, Monday as first day of the week (00–53)
%x	local date representation
%X	local time representation
%y	year without century (00–99)
%Y	year with century
%Z	time zone name, if any
%%	%

2.2.24 unistd.h

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

`access(*name,mode)`	Use the file system simulation of the debugger to check the permissions of a file on the host. *mode* specifies the type of access and is a bit pattern constructed by a logical OR of the following values: `R_OK` Checks read permission. `W_OK` Checks write permission. `X_OK` Checks execute (search) permission. `F_OK` Checks to see if the file exists. *(FSS implementation)*
`chdir(*path)`	Use the file system simulation feature of the debugger to change the current directory on the host to the directory indicated by *path*. *(FSS implementation)*
`close(fd)`	File close function. The given file descriptor should be properly closed. This function calls `_close()`. *(FSS implementation)*
`getcwd(*buf,size)`	Use the file system simulation feature of the debugger to retrieve the current directory on the host. Returns the directory name. *(FSS implementation)*
`lseek(fd,offset, whence)`	Moves read–write file offset. Calls `_lseek()`. *(FSS implementation)*
`read(fd,*buff,cnt)`	Reads a sequence of characters from a file. This function calls `_read()`. *(FSS implementation)*
`stat(*name,*buff)`	Use the file system simulation feature of the debugger to stat() a file on the host platform. *(FSS implementation)*
`unlink(*name)`	Removes the named file, so that a subsequent attempt to open it fails. Calls `_unlink()`. *(FSS implementation)*
`write(fd,*buff,cnt)`	Write a sequence of characters to a file. Calls `_write()`. *(FSS implementation)*

2.2.25 wchar.h

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

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

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

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

`mbsinit(*ps)`	Determines whether the object pointed to by *ps*, is an initial conversion state. Returns a non-zero value if so.
`mbsrtowcs(*pwcs,**src,n,*ps)`	Restartable version of `mbstowcs`. See section 2.2.21, *stdlib.h*. The initial conversion state is specified by *ps*. The input sequence of multibyte charactersis specified indirectly by *src*.
`wcsrtombs(*s,**src,n,*ps)`	Restartable version of `wcstombs`. See section 2.2.21, *stdlib.h*. The initial conversion state is specified by *ps*. The input wide string is specified indirectly by *src*.
`mbrtowc(*pwc,*s,n,*ps)`	Converts a multibyte character **s* to a wide character **pwc* according to conversion state *ps*. See also `mbtowc` in section 2.2.21, *stdlib*.
`wcrtomb(*s,wc,*ps)`	Converts a wide character wc to a multi-byte character according to conversion state *ps* and stores the multi-byte character in **s*.
`btowc(c)`	Returns the wide character corresponding to character *c*. Returns WEOF on error.
`wctob(c)`	Returns the multi-byte character corresponding to the wide character *c*. The returned multi-byte character is represented as one byte. Returns EOF on error.
`mbrlen(*s,n,*ps)`	Inspects up to *n* bytes from the string **s* to see if those characters represent valid multibyte characters, relative to the conversion state held in **ps*.

2.2.26 wctype.h

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

`wctype(*property)`	Constructs a value of type `wctype_t` that describes a class of wide characters identified by the string **property*. If property identifies a valid class of wide characters according to the LC_TYPE category (see 2.2.12, *locale.h*) of the current locale, a non–zero value is returned that can be used as an argument in the `iswctype` function.
`iswctype(wc,desc)`	Tests whether the wide character *wc* is a member of the class represented by `wctype_t` *desc*. Returns a non–zero value if tested true.

Function	Equivalent to locale specific test
`iswalnum(wc)`	`iswctype(wc,wctype("alnum"))`
`iswalpha(wc)`	`iswctype(wc,wctype("alpha"))`
`iswcntrl(wc)`	`iswctype(wc,wctype("cntrl"))`
`iswdigit(wc)`	`iswctype(wc,wctype("digit"))`
`iswgraph(wc)`	`iswctype(wc,wctype("graph"))`
`iswlower(wc)`	`iswctype(wc,wctype("lower"))`
`iswprint(wc)`	`iswctype(wc,wctype("print"))`
`iswpunct(wc)`	`iswctype(wc,wctype("punct"))`
`iswspace(wc)`	`iswctype(wc,wctype("space"))`
`iswupper(wc)`	`iswctype(wc,wctype("upper"))`
`iswxditig(wc)`	`iswctype(wc,wctype("xdigit"))`

`wctrans(*property)`	Constructs a value of type `wctype_t` that describes a mapping between wide characters identified by the string **property*. If property identifies a valid mapping of wide characters according to the LC_TYPE category (see 2.2.12, *locale.h*) of the current locale, a non–zero value is returned that can be used as an argument in the `towctrans` function.
`towctrans(wc,desc)`	Transforms wide character *wc* into another wide–character, described by *desc*.

Function	Equivalent to locale specific transformation
`towlower(wc)`	`towctrans(wc,wctrans("tolower")`
`towupper(wc)`	`towctrans(wc,wctrans("toupper")`

3 Assembly Language

Summary

This chapter contains a detailed description of all built–in assembly functions and assembler directives. For a description of the assembly instruction set, refer to the core reference manual of the relevant target.

3.1 Built–in Assembly Functions

3.1.1 Overview of Built–in Assembly Functions

The following table provides an overview of all built–in assembly functions. Next all functions are described into more detail. *expr* can be any assembly expression resulting in an integer value. Expressions are explained in section 3.6, *Assembly Expressions*, in the user's manual.

Overview of assembly functions

Function	Description
`@ARG('`*symbol*`'` | *expr*`)`	Test whether macro argument is present
`@BIGENDIAN()`	Test if assembler generates code for big–endian mode
`@CNT()`	Return number of macro arguments
`@DEFINED('`*symbol*`'` | *symbol*`)`	Test whether *symbol* exists
`@GPREL(`*symbol*`)`	Offset of *symbol* from the global pointer (R28)
`@HI(`*expr*`)`	Most significant half word of the expression, sign adjusted
`@LO(`*expr*`)`	Least significant half word of the expression, sign adjusted
`@LSB(`*expr*`)`	Least significant byte of the expression
`@LSH(`*expr*`)`	Least significant half word of the expression
`@MSB(`*expr*`)`	Most significant byte of the expression
`@MSH(`*expr*`)`	Most significant half word of the expression
`@STRCAT(`*str1*`,`*str2*`)`	Concatenate *str1* and *str2*
`@STRCMP(`*str1*`,`*str2*`)`	Compare *str1* with *str2*

Function	Description
`@STRLEN(`*`str`*`)`	Return length of string
`@STRPOS(`*`str1`*`,`*`str2`*`[,`*`start`*`])`	Return position of *str1* in *str2*

3.1.2 Detailed Description of Built-in Assembly Functions

@ARG('symbol' | expression)

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

You can specify the argument with a *symbol* name (the name of a macro argument enclosed in single quotes) or with *expression* (the ordinal number of the argument in the macro formal argument list).

If you use this function when macro expansion is not active, the assembler issues a warning.

Example:

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

@BIGENDIAN()

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

@CNT()

Returns the number of macro arguments of the current macro expansion as an integer.

If you use this function when macro expansion is not active, the assembler issues a warning.

Example:

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

@DEFINED('symbol' | symbol)

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

Example:

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

@GPREL(symbol)

Returns the offset of *symbol* from the global pointer ($28).

If you want the assembler to generate GP-relative offsets automatically (option **--gp-relative**) enable the option **Automatically generate GP-relative offsets** in the **Assembler Miscellaneous** page of the **Project Options** dialog.

@HI(expression)

Returns the *most* significant half word of the result of the *expression*, adjusted for signed addition. `@HI(expression)` is equivalent to `((expression>> 16) + ((expression & 0x8000) ? 1 : 0)) & 0xFFFF`. *expression* can be any relocatable or absolute expression.

Example:

```
; The instruction  lw $2, addr expands to
lui at, @hi(addr)
lw  $2, @lo(addr)(at)
```

@LO(expression)

Returns the *least* significant half word (bits 0..15) of the result of the *expression*, adjusted for signed addition. *expression* can be any relocatable or absolute expression.

@LSB(expression)

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

@LSH(expression)

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

@MSB(expression)

Returns the *most* significant byte of the result of the *expression*.
The result of the expression is calculated as16 bits.

@MSH(expression)

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

@STRCAT(string1,string2)

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

Example:

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

@STRCMP(*string1,string2****)***

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

`<0`	if *string1* < *string2*
`0`	if *string1* == *string2*
`>0`	if *string1* > *string2*

Example:

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

@STRLEN(*string****)***

Returns the length of *string* as an integer.

Example:

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

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

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

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

Example:

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

3.2 Assembler Directives

3.2.1 Overview of Assembler Directives

Assembler directives are grouped in the following categories:

- Assembly control directives
- Symbol definition directives
- Data definition / Storage allocation directives
- Macro and conditional assembly directives
- Listing control and options directives
- HLL directives

The following tables provide an overview of all assembler directives.

Overview of assembly control directives

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

Overview of symbol definition directives

Directive	Description
`.EQU`	Set permanent value to a symbol
`.EXTERN`	Import global section symbol
`.GLOBAL`	Declare global section symbol
`.RESUME`	Resume a previously defined section
`.SECTION/.ENDSEC`	Start a new section
`.SET`	Set temporary value to a symbol
`.SIZE`	Set size of symbol in the ELF symbol table
`.SOURCE`	Specify name of original C source file
`.TYPE`	Set symbol type in the ELF symbol table
`.WEAK`	Mark a symbol as ’weak’

Overview of data definition / storage allocation directives

Directive	Description
`.ALIGN`	Align location counter
`.BS/.BSB/.BSH/.BSW`	Define block storage (initialized)
`.DB`	Define byte
`.DH`	Define half word
`.DW`	Define word
`.DS/.DSB/.DSH/.DSW`	Define storage
`.OFFSET`	Move location counter forwards

Overview of macro and conditional assembly directives

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

Overview of listing control assembly directives

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

Overview of HLL directives

Directive	Description
`.CALLS`	Pass call tree information

TSK3000 specific directive

Directive	Description
`.NOPINSERTION`	Insert a NOP instruction after jump and branch instructions
`.NONOPINSERTION`	No extra NOP instruction after jump and branch instructions

3.2.2 Detailed Description of Assembler Directives

Each assembler directive has its own syntax. Some assembler directives can be preceeded with a label. If you do not preceede an assembler directive with a label, you must use white space instead (spaces or tabs). You can use assembler directives in the assembly code as pseudo instructions.

.ALIGN

Syntax

.ALIGN *expression*

Description

With the `.ALIGN` directive you tell the assembler to align the location counter.

When the assembler encounters the `.ALIGN` directive, it moves the location counter forwards 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. If the location counter is already aligned on the specified alignment, it remains unchanged. The location of absolute sections will not be changed.

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

Examples

```
.SECTION .text
.ALIGN 16     ; the assembler aligns
instruction   ; this instruction at 16 MAUs and
              ; fills the ’gap’ with NOP instructions.

.SECTION .text
.ALIGN 12     ; WRONG: not a power of two, the
instruction   ; assembler aligns this instruction at
              ; 16 MAUs and issues a warning.
```

.BREAK

Syntax

.BREAK

Description

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

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

Example

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

.BS/.BSB/.BSH/.BSW

Syntax

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

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

[*label*] .**BSH** *expression1*[,*expression2*]

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

Description

With the `.BS` directive (Block Storage) the assembler reserves a block of memory. The reserved block of memory is initialized to the value of *expression2*, or zero if omitted.

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

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

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

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

The `.BSB`, `.BSH` and `.BSW` directives are variants of the `.BS` directive:

.BSB	The *expression1* argument specifies the number of bytes to reserve.
.BSH	The *expression1* argument specifies the number of half words to reserve (one half word is16 bits).
.BSW	The *expression1* argument specifies the number of words to reserve (one word is 32 bits).

Example

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

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

Related information

.DS (Define Storage)

.CALLS

Syntax

.CALLS '*caller*', '*callee*'

Description

With this directive you indicate that a function *caller* calls another function *callee*.

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

The linker uses the `.CALLS` information to build a call graph.

Example

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

Indicates that the function `main` calls the function `nfunc`

.DB

Syntax

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

Description

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

An *argument* can be:

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

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

Multiple arguments are stored in successive address locations. If an argument is NULL, its corresponding address location is flled with zeros.

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

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

```
   .DB 'R'         ; = 0x52
   .DB 'AB',,'D'   ; = 0x41420043  (second argument is empty)
```

Example

```
TABLE:   .DB 14,253,0x62,'ABCD'
CHARS:   .DB 'A','B',,'C','D'
```

Related information

.BS (Block Storage)
.DS (Define Storage)
.DH (Define Half Word)
.DW (Define Word)

.DEFINE

Syntax

.**DEFINE** *symbol string*

Description

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

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

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

Example

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

```
.DEFINE LEN ”32”
```

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

```
.DS LEN
.MESSAGE I ”The length is: LEN”
```

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

```
.DS 32
.MESSAGE I ”The length is: 32”
```

Related information

.UNDEF (Undefine a .DEFINE symbol or macro)
.MACRO/**.ENDM** (Define a macro)

.DH

Syntax

[*label*] **.DH** *argument*[,*argument*]...

Description

With the `.DH` directive (Define Half Word) you allocate and initialize a half word of memory for each *argument*.

A half word is 16 bits.

An *argument* is:

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

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

Multiple arguments are stored in successive half word address locations. If an argument is NULL, its corresponding address location is filled with zeros.

Half word arguments are stored as is. Floating-point values are not allowed.

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

In case of single and multiple character strings, each character is stored in the least significant byte of a half word whose lower seven bits represent the ASCII value of the character. The standard C escape sequences are allowed:

```
.DH 'AB',,'D' => 0x0041
                 0x0042
                 0x0000 (second argument is empty)
                 0x0044
```

Example

```
TABLE:   .DH 14,253,0x62,'ABCD'
CHARS:   .DH 'A','B',,'C','D'
```

Related information

.BS (Block Storage)
.DS (Define Storage)
.DB (Define Byte)
.DW (Define Word)

.DS/.DSB/.DSH/.DSW

Syntax

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

[*label*] **.DSB** *expression*

[*label*] **.DSH** *expression*

[*label*] **.DSW** *expression*

Description

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

With the *expression* you specify the number of minimum addressable units (MAUs) that you want to reserve. The expression must evaluate to an integer larger than zero and cannot contain references to symbols that are not yet defined in the assembly source.

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

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

The `.DSB`, `.DSH` and `.DSW` directives are variants of the `.DS` directive:

.DSB	The *expression* argument specifies the number of bytes to reserve.
.DSH	The *expression* argument specifies the number of half words to reserve (one half word is16 bits).
.DSW	The *expression* argument specifies the number of words to reserve (one word is 32 bits).

Example

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

Related information

.BS (Block Storage)
.DB (Define Byte)
.DH (Define Half Word)
.DW (Define Word)

.DW

Syntax

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

Description

With the `.DW` directive (Define Word) you allocate and initialize one word of memory for each *argument*.

One word is 32 bits.

An *argument* is:

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

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

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

Word arguments are stored as is. Floating–point values are not allowed. If the evaluated argument is too large to be represented in a word, the assembler issues an error and truncates the value.

In case of character strings, each character is stored in the least significant byte of a word which represents the ASCII value of the character:

```
   .DW 'AB',,'D' => 0x0000041
                    0x0000042
                    0x0000000 (second argument is empty)
                    0x0000044
```

Example

```
TABLE:  .DW 14,253,0x62,'ABCD'
CHARS:  .DW 'A','B',,'C','D'
```

Related information

.BS (Block Storage)
.DS (Define Storage)
.DB (Define Byte)
.DH (Define Half Word)

.END

Syntax

.END

Description

With the `.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.

Example

```
.SECTION     code, code
mov  R7,#9
mov  R6,#0
ret
.END                    ; End of assembly module
```

.EQU

Syntax

symbol **.EQU** *expression*

Description

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

Example

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

```
MYSYMBOL .EQU  0x4000
```

You cannot redefine the used symbols.

Related information

.SET (Set temporary value to a symbol)

.EXTERN

Syntax

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

Description

With the `.EXTERN` directive you define an *external* symbol. It means that the symbol is referenced in the current module while it is defined outside the current module.

You must define the symbols either outside any module or declare it 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.

Example

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

Related information

.GLOBAL (Declare global section symbol)

.FOR/.ENDFOR

Syntax

[*label*] **.FOR** *var* **IN** *expression*[,*expression*]...
    ....
    .ENDFOR

or:

[*label*] **.FOR** *var* **IN** *start* **TO** *end* [**STEP** *step*]
    ....
    .ENDFOR

Description

With the `.FOR/.ENDFOR` directive you can repeat a sequence of assembly source lines with an iterator. As shown by the syntax, you can use the `.FOR/.ENDFOR` in two ways.

1. In the first mehod, the loop is repeated as many times as the number of arguments following `IN`. If you use the symbol *var* in the assembly lines between `.FOR` and `.ENDFOR`, for each repetition the symbol *var* is substituted by a subsequent *expression* from the argument list. If the argument is a null, then the loop is repeated with each occurrence of the symbol *var* removed.
2. In the second method, the loop is repeated using the symbol *var* as a counter. The counter passes all integer values from *start* to *end* with a *step*. If you do not specify *step*, the counter is increased by one for every repetition.

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

Example

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

```
.FOR VAR1 IN 1,2+3,4,12
     .DB VAR1
     .DW (VAR1*VAR1)
.ENDFOR
```

In the following example the loop is repeated 16 times. With the `.DW` directive you allocate and initialize four bytes of memory for each repetition of the loop. Effectively, the preprocessor duplicates the `.DW` directive16 times in the assembled file, and substitutes `VAR2` with the subsequent numbers.

```
.FOR VAR2 IN 1 to 0x10
     .DW (VAR1*VAR1)
.ENDFOR
```

Related information

.REPEAT/.ENDREP (Repeat sequence of source lines)

.GLOBAL

Syntax

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

Description

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

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.

Example

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

Related information

.EXTERN (Import global section symbol)

.IF/.ELIF/.ELSE/.ENDIF

Syntax

.IF *expression*	
.	
.	
[**.ELIF** *expression*]	(the `.ELIF` directive is optional)
.	
.	
[**.ELSE**]	(the `.ELSE` directive is optional)
.	
.	
.ENDIF	

Description

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

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

You can nest `.IF` directives to any level. The `.ELSE` and `.ELIF` directive always refer to the nearest previous `.IF` 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 in Altium Designer as preprocessor macros in dialog **Project » Project Options » Assembler » Preprocessing** (assembler option **–D**).

Related information

Assembler option **-D** (Define preprocessor macro) in Section 4.2, *Assembler Options*, of Chapter *Tool Options*.

.INCLUDE

Syntax

.INCLUDE ”*filename*” | <*filename*>

Description

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

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

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

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

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

2. The path that is specified with the assembler option **–I**.

3. The path that is specified in the environment variable AS*target*INC when the product was installed.

4. The default directory `...\`*`ctarget`*`\include`.

Example

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

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

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

```
.INCLUDE ”myinc.inc”
```

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

Related information

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

.LIST/.NOLIST

Syntax

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

Description

If you generate a list file (see assembler option **–l**), you can use the `.LIST` and `.NOLIST` directives to specify which source lines the assembler must write to the list file.

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

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

Example

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

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

The assembler generates a list file with the following lines:

```
.SECTION .text
...  ; source line 1
.LIST
...  ; source line 3
.END
```

Related information

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

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

Example

The macro definition:

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

The macro call:

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

The macro expands as follows:

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

Related information

.DEFINE (Define a substitution string)
Section 3.9, *Macro Operations*, in Chapter *Assembly Language* of the user's manual.

.MESSAGE

Syntax

.MESSAGE *type* [{*str*|*exp*|*symbol*}[,{*str*|*exp*|*symbol*}]...]

Description

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

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

I Information message. Error and warning counts are not affected and the assembler continues the assembling process.

W Warning message. Increments the warning count and the assembler continues the assembling process.

E Error message. Increments the error count and the assembler continues the assembling process.

F Fatal error message. The assembler immediately aborts the assembling process and generates no object file or list file.

The `.MESSAGE` directive is for example useful in combination with conditional assembly to indicate which part is assembled.

Example

```
   .MESSAGE I 'Generating tables'

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

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

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

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

.NOPINSERTION/.NONOPINSERTION

Syntax

.NOPINSERTION
.
. ; assembly source lines
.
.NONOPINSERTION

Description

You can instruct the assembler to automatically fill the delay slots of jump and branch instructions with a NOP instruction (see assembler option **––nop–insertion**). With the `.NOPINSERTION` and `.NONOPINSERTION` directives you have more control over te NOP insertion.

Example

```
.section .text

.nopinsertion
  jr      $2        ; a nop is added after each instruction
  jalr    $2,$3
.nonopinsertion
  jr      $2        ; no extra nop instruction is added
  jalr    $2,$3
```

Related information

Assembler option **––nop–insertion** in Section 4.2, *Assembler Options*, of Chapter *Tool Options*.

.OFFSET

Syntax

.OFFSET *expression*

Description

With the `.OFFSET` directive you tell the assembler to give the location counter a new offset relative to the start of the section.

When the assembler encounters the `.OFFSET` directive, it moves the location counter forwards to the specified address, relative to the start of the section, and places the next instruction on that address. If you specify an address equal to or lower than the current position of the location counter, the assembler issues an error.

Example

```
.SECTION .text
nop
nop
nop
.OFFSET 0x20    ; the assembler places
nop             ; this instruction at address 0x20
                ; relative to the start of the section.

.SECTION .text
nop
nop
nop
.OFFSET 0x02    ; WRONG: the current position of the
nop             ; location counter is 0x0C.
```

.PAGE

Syntax

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

Description

If you generate a list file (see assembler option **-l**), you can use the `.PAGE` directive to format the generated list file.

width	Number of characters on a line (1–255). Default is 132.
length	Number of lines per page (10–255). Default is 66.
blanktop	Number of blank lines at the top of the page. Default = 0. Specify a value so that *blanktop* + *blankbtm* ≤ *length* – 10.
blankbtm	Number of blank lines at the bottom of the page. Default = 0. Specify a value so that *blanktop* + *blankbtm* ≤ *length* – 10.
blankleft	Number of blank columns at the left of the page. Default = 0. Specify a value smaller than *width*.

If you use the `.PAGE` directive without arguments, it causes a 'formfeed': the next source line is printed on the next page in the list file.

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

A label is not allowed with this directive.

Example

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

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

.PAGE ,,5   ; insert five blank lines at the top. Note
            ; that you can omit the last two arguments.
```

Related information

.TITLE (Set program title in header of assembler list file)
Assembler option **-l** (Generate list file) in Section 4.2, *Assembler Options*, of Chapter *Tool Options*.

.REPEAT/.ENDREP

Syntax

[*label*] **.REPEAT** *expression*

    ....
    .ENDREP

Description

With the `.REPEAT/.ENDREP` directive you can repeat a sequence of assembly source lines. With *expression* you specify the number of times the loop is repeated.

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

Example

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

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

Related information

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

.RESUME

Syntax

.RESUME *name* [, *attribute*]...

Description

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

Example

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

Related information

.SECTION (Start a new section)

.SECTION

Syntax

.SECTION *name* [**,at(***address***)**]
....
[**.ENDSEC**]

Description

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

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

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

You can use the following predefined section names:

Section Name	Description	Section Type
.text	Code sections	code
.data	Initialized data	data
.sdata	Initialized data in read−write small data area	data
.bss	Uninitialized data (cleared)	data
.sbss	Uninitialized data in read−write small data area (cleared)	data
.rodata	ROM data (constants)	data
.debug	Debug sections	debug

Table 3−1: Predefined section names

Sections of a specified type are located by the linker in a memory space. The space names are defined in a so−called 'linker script file' (files with the extension `.lsl`) delivered with the product in the directory `\Program Files\Altium2004\System\Tasking\include.lsl`.

You can specify the following section attributes:

Example

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

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

.RESUME (Resume a previously defined section)

.SET

Syntax

symbol **.SET** *expression*

.SET *symbol expression*

Description

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

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

Example

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

Related information

.EQU (Set a permanent value to a symbol)

.SIZE

Syntax

.SIZE *symbol*, *expression*

Description

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

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

Example

```
        .section        .text
        .align  4
        .global main
; Function main
main:   .type   func
         ;
        .SIZE   main,*-main
        .endsec
```

Related information

.TYPE (Set Symbol Type)

.SOURCE

Syntax

.SOURCE *string*

Description

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

Example

```
.SOURCE ”test.c”
```

Related information

–

.TITLE

Syntax

.TITLE [*title*]

Description

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

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

Example

```
.TITLE "The best program"
```

In the header of each page in the assembler list file, the title of the progam is printed. In this case: `The best program`

Related information

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

.TYPE

Syntax

symbol **.TYPE** *typeid*

Description

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

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

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

Example

```
Afunc:   .TYPE   FUNC
```

Related information

.SIZE (Set Symbol Size)

.UNDEF

Syntax

.UNDEF *symbol*

Description

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

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

Example

```
.UNDEF LEN
```

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

Related information

.DEFINE (Define substitution string)

.WEAK

Syntax

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

Description

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

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

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

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

Example

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

Related information

.EXTERN (Import global section symbol)
.GLOBAL (Declare global section symbol)

3.3 Generic Instructions

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

You can find a complete list of generic instructions for the TSK3000 in the core reference manual *CR0121 TSK3000A 32−bit RISC Processor*.

4 Tool Options

Summary

This chapter provides a detailed description of the options for the compiler, assembler, linker, control program, make program and the librarian.

4.1 Compiler Options

Altium Designer uses a makefile to build your entire project. This means that in Altium Designer you cannot run the compiler separately. If you compile a single C source file from within Altium Designer, the file is also assembled. However, you can set options specific for the compiler.

Options in Altium Designer versus options on the command line

Most command line options have an equivalent option in Altium Designer but some options are only available on the command line (for example in a Windows Command Prompt). If there is no equivalent option in Altium Designer, you can specify a command line option in Altium Designer as follows:

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

   The Project Options dialog box appears.

2. Expand the **C Compiler** entry and select **Miscellaneous**.
3. Enter one or more command line options in the **Additional C Compiler options** field.

Invocation syntax on the command line (Windows Command Prompt)

To call the compiler from the command line, use the following syntax:

```
c3000 [ [option]... [file]... ]...
```

The input *file* must be a C source file (`.c` or `.ic`).

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 double minus (––) characters. You can abbreviate long option names as long as the name is unique. You can mix short and long option names on the command line.

Options can have flags or sub−options. 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:

```
c3000 -Oac test.c
```

```
c3000 --optimize=+coalesce,+cse test.c
```

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

Compiler: –? (––help)

Menu entry

Command line only.

Command line syntax

–?
––help[=*item*,...]

You can specify the following arguments:

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

Description

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

Example

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

```
c3000 -?
c3000 --help
c3000
```

The following invocation displays a list of the available pragmas:

```
c3000 --help=pragmas
```

Compiler: –A (––language)

Menu entry

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

   The Project Options dialog box appears.

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

3. Enable or disable the following options:

   - **Allow C++ style comments in C source code** (only available when **ISO C 90** is selected)
   - **Relax const check for string literals**

Command line syntax

–A[*flags*]
––language=[*flags*]

You can set the following flags:

p/**P**	(+/–**comments**)	Allow C++ style comments in C source code
x/**X**	(+/–**strings**)	Relaxed const check for string literals

–A (**––language**) is the equivalent of **–APX** which disables all language extensions.
The default is **–Apx**.

Description

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

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

With **Relax const check for string literals** (**–Ax**) you tell the compiler not to check for assignments of a constant string to a non–constant string pointer. With this option the following example produces no warning:

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

Example

```
c3000 -APx -c90 test.c
c3000 --language=-comments,+strings --iso=90 test.c
```

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

Compiler: –c (––iso)

Menu Entry

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

   The Project Options dialog box appears.

2. Expand the **C Compiler** entry and select **Language**.
3. Select the ISO C standard **C90** or **C99**.

Command line syntax

–c{**90**|**99**}
––iso={**90**|**99**}

Description

With this option you select the ISO C standard. The compiler checks the C source against this standard and may generate warnings or errors if you use C language that is not defined in the standard.

C90 is also referred to as the ”ANSI C standard”. C99 refers to the newer ISO/IEC 9899:1999 (E) standard. C99 is the default.

```
c3000 −c90 test.c
c3000 −−iso=90 test.c
```

Related information

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

Compiler: −−check

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **C Compiler** entry and select **Miscellaneous**.
3. Add the option **−−check** to the **Additional C compiler options** field.

Command line syntax

−−check

Description

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

The compiler reports any warnings and/or errors.

Related information

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

Compiler: –D (––define)

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **C Compiler** entry and select **Preprocessing**.
3. Next to **User macro**, click on the down arrow in the right pane to expand macro input.
4. Click on an empty **Macro** field and enter a macro name. (Then click an empty cell to confirm)
5. Optionally, click in the **Value** field and enter a definition. (Then click an empty cell to confirm)

Command line syntax

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

Description

With this option you can define a macro and specify it to the preprocessor. If you only specify a macro name (no macro definition), the macro expands as '1'. You can specify as many macros as you like.

On the command line, you can use the option **–D** multiple times. If the command line exceeds the length limit of the operating system, you can define the macros in an *option file* which you then must specify to the compiler with the option **–f** *file*.

Defining macros with this option (instead of in the C source) is, for example, useful to compile conditional C source as shown in the example below.

Example

Consider the following C program with conditional code to compile a demo program and a real program:

```
void main( void )
{
#if DEMO == 1
    demo_func();   /* compile for the demo program */
#else
    real_func();   /* compile for the real program */
#endif
}
```

You can now use a macro definition to set the DEMO flag:

Macro	Value
DEMO	1 (or empty)

On the command line, use the option −**D** as follows:

```
c3000 -DDEMO test.c
c3000 -DDEMO=1 test.c

c3000 --define=DEMO test.c
c3000 --define=DEMO=1 test.c
```

Note that all four invocations have the same effect.

The next example shows how to specify a macro with arguments. Macro definitions follow exactly the same rules as the `#define` statement in the C language.

Macro	Value
`MAX(A,B)`	`((A) > (B) ? (A) : (B))`

On the command line, use the option −**D** as follows:

```
c3000 -D"MAX(A,B)=((A) > (B) ? (A) : (B))" test.c
```

Note that the macro name and definition are placed between double quotes because otherwise the spaces would indicate a new option.

Related information

Compiler option **–U** (Undefine preprocessor macro)
Compiler option **–f** (Read options from file)

Compiler: ––diag

Menu entry

1. From the **View** menu, select **Workspace » Panels » System Messages**.

   The Message pannel appears.

2. In the **Message** panel, right–click on the message you want more information on.

   A popup menu appears.

3. Select **More Info**.

   A Message Info box appears with additional information.

Command line syntax

––diag=[*format*:]{**all**|*nr*,...]

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to `stdout` (normally your screen) and in the format you specify. You can specify the following formats: **html**, **rtf** or **text** (default). To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas.

With this option the compiler does not compile any files.

Example

To display an explanation of message number 282, enter:

```
c3000 --diag=282
```

This results in the following message and explanation:

```
E282: unterminated comment

Make sure that every comment starting with /* has a matching */. Nested
comments are not possible.
```

To write an explanation of all errors and warnings in HTML format to file `cerrors.html`, use redirection and enter:

```
c3000 --diag=html:all > cerrors.html
```

Related information

–

Compiler: –E (––preprocess)

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **C Compiler** entry and select **Preprocessing**.
3. Enable the option **Store the C Compiler preprocess output (<file>.pre)**.

Command line syntax

–E[*flags*]
––preprocess[=*flags*]

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

c/**C** (+/–**comments**)	Keep comments from the C source in the preprocessed output
m/**M** (+/–**make**)	Generate dependency lines that can be used for the makefile
p/**P** (+/–**noline**)	Strip #line source position information (lines starting with `#line`)

The compiler sends the preprocessed file to stdout. To capture the information in a file, specify an output file with the option **–o**.

Description

When compiling, each file is preprocessed first. With this option you can store the result of preprocessed C files. Altium Designer stores the preprocessed file in a file called *name*.pre (where *name* is the name of the C source file being compiled). C comments are not preserved (similar to **–ECMP**)

Related information

–

Compiler: ––error–file

Menu entry

Command line only.

Command line syntax

––error–file[=*file*]

Description

With this option the compiler redirects error messages to a file.

If you do not specify a filename, the error file will be named after the input file with extension `.err`.

Example

To write errors to `errors.err` instead of `stderr`, enter:

```
c3000 --error-file=errors.err test.c
```

Related information

–

Compiler: –F (––no–double)

Menu entry

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

   The Project Options dialog box appears.

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

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

Command line syntax

–F

––no–double

Description

With this option you tell the compiler to treat variables of the type `double` as `float`. Because the float type takes less space, execution speed increases and code size decreases, both at the cost of less precision.

Related information

–

Compiler: –f (––option–file)

Menu entry

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

   The Project Options dialog box appears.

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

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

Be aware that when you specify the option **–f** in the **Additional C compiler options** field, the options are added to the compiler options you have set in the other dialogs. Only in extraordinary cases you may want to use them in combination. Altium Designer automatically saves the options with your project.

Command line syntax

–f *file*
––option–file=*file*

Description

This option is primarily intended for command line use. Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the compiler.

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

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

Format of an option file

- Multiple arguments on one line in the option file are allowed.
- To include whitespace in an argument, surround the argument with single or double quotes.
- If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

```
"This has a single quote ' embedded"

'This has a double quote " embedded'

'This has a double quote " and a single quote '"' embedded"
```

- When a text line reaches its length limit, use a \ to continue the line. Whitespace between quotes is preserved.

```
"This is a continuation \
line"

    -> "This is a continuation line"
```

- It is possible to nest command line files up to 25 levels.

Example

Suppose the file `myoptions` contains the following lines:

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

Specify the option file to the compiler:

```
c3000 -f myoptions
c3000 --option-file=myoptions
```

This is equivalent to the following command line:

```
c3000 -g -DDEMO=1 test.c
```

Related information

-

Compiler: –g (––debug–info)

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **C Compiler** entry and select **Debug Information**.
3. Enable the option **Generate symbolic debug information**.
4. Enable or disable the suboptions.

Command line syntax

–g[**c**|**a**]

––debug–info[=*suboption*]

You can set the following suboptions (when you specify **–g** without suboption, the default is **–ga**):

c	(**call–frame**)	Generate call–frame information only.
a	(**all**)	Generate all debug information.

Description

With this option you tell the compiler to add directives to the output file for including symbolic information. This facilitates high level debugging but increases the size of the resulting assembler file (and thus the size of the object file). For the final application, compile your C files without debug information.

When you specify a high optimization level, debug comfort may decrease. Therefore, the compiler issues a warning if the chosen optimizations expect to affect ease of debugging.

call–frame information

With this suboption only call–frame information is generated. This enables you to inspect parameters of nested functions.

default debug information

This provides all debug information you need to debug your application. It meets the debugging requirements in most cases without resulting in over–sized assembler/object files.

all debug information

With this information extra debug information is generated. In extra–ordinary cases you may use this debug information (for instance, if you use your own debugger which makes use of this information). With this suboption, the resulting assembler/object file increases significantly.

Related information

–

Compiler: –H (––include–file)

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **C Compiler** entry and select **Preprocessing**.
3. Enter the name of the file in the **Include this file before source** field or click **...** and select a file.

Command line syntax

–H*file*,...
––include–file=*file*,...

Description

With this option (set at project level) you include one extra file at the beginning of each C source file in your project. On a document level (**Project** » **Document Options**), you can overrule this option with another file or no file at all.

The specified include file is included before all other includes. This is the same as specifying `#include "file"` at the very beginning of (each of) your C source files.

Example

```
c3000 -Hstdio.h test1.c test2.c
c3000 --include-file=stdio.h test1.c test2.c
```

The file `stdio.h` is included at the beginning of both `test1.c` and `test2.c`.

Related information

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

Section 4.5, *How the Compiler Searches Include Files*, in chapter *Using the Compiler* of the user's manual.

Compiler: –I (––include–directory)

Menu entry

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

   The Project Options dialog box appears.

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

3. Add a pathname in the **Include files path** field.

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

Command line syntax

–I*path*,...
––include–directory=*path*,...

Description

With this option you can specify the path where your include files are located. A relative path will be relative to the current directory.

The order in which the compiler searches for include files is:

1. The pathname in the C source file and the directory of the C source (only for #include files that are enclosed in "").

2. The path that is specified with this option.

3. The path that is specified in the environment variable `C3000INC` when the product was installed.

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

Example

Suppose that the C source file `test.c` contains the following lines:

```
#include <stdio.h>
#include "myinc.h"
```

You can specify the include directory `myinclude` to the compiler:

```
c3000 –Imyinclude test.c
c3000 ––include–directory=myinclude test.c
```

First the compiler looks for the file `stdio.h` in the directory `myinclude` relative to the current directory. If it was not found, the compiler searches in the environment variable and then in the default `include` directory.

The compiler now looks for the file `myinc.h`, in the directory where `test.c` is located. If the file is not there the compiler searches in the directory `myinclude`. If it was still not found, the compiler searches in the environment variable and then in the default `include` directory.

Related information

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

Compiler: ––inline

Menu entry

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

   The Project Options dialog box appears.

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

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

Command line syntax

––inline

Description

With this option you instruct the compiler to inline all functions, regardless whether they have the keyword `inline` or not. This option has the same effect as a `#pragma inline` at the start of the source file.

This option can be useful to increase the possibilities for code compaction (compiler option **–Or**).

Related information

Compiler option **–Or** (Code compaction)

Compiler: ––inline–max–incr / ––inline–max–size

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **C Compiler** entry and select **Optimization**.
3. Set the option **Maximum code size increase caused by inlining** to a value (default: 25)
4. Set the option **Maximum size for functions to always inline** to a value (default: 25)

Command line syntax

––inline–max–incr=*percentage*	(Default: 25)
––inline–max–size=*threshold*	(Default: 25)

Description

With these options you can control the function inlining optimization process of the compiler. These options have only effect when you have enabled the inlining optimization (option **–Oi**).

Regardless of the optimization process, the compiler always inlines all functions that have the function qualifier `inline`.

With the option **––inline–max–size** you can specify the maximum size of functions that the compiler inlines as part of the optimization process. The compiler always inlines all functions that are smaller than the specified threshold. The threshold is measured in compiler internal units and the compiler uses this measure to decide which functions are small enough to inline.

After the compiler has inlined all functions that have the function qualifier inline and all functions that are smaller than the specified threshold, the compiler looks whether it can inline more functions without increasing the code size too much. With the option **––inline–max–incr** you can specify how much the code size is allowed to increase. By default, this is 25% which means that the compiler continues inlining functions until the resulting code size is 25% larger than the original size.

Example

```
c3000 --inline-max-incr=40 --inline-max-size=15 test.c
```

The compiler first inlines all functions with the function qualifier inline and all functions that are smaller than the specified threshold of 15. If the code size has still not increased with 40%, the compiler decides which other functions it can inline.

Related information

Compiler option –O (Specify optimization level)

Section 2.7.3, *Inlining Functions*, in chapter *C Language* of the user's manual.

Compiler: –k (––keep–output–files)

Menu entry

Altium Designer *always* removes the `.src` file when errors occur during compilation.

Command line syntax

–k
––keep–output–files

Description

If an error occurs during compilation, the resulting `.src` file may be incomplete or incorrect. With this option you keep the generated output file (`.src`) when an error occurs.

By default the compiler removes the generated output file (`.src`) when an error occurs. This is useful when you use the make utility. If the erroneous files are not removed, the make utility may process corrupt files on a subsequent invocation.

Use this option when you still want to inspect the generated assembly source. Even if it is incomplete or incorrect.

Related information

–

Compiler: –m (––call)

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **C Compiler** entry and select **Code Generation**.
3. Set the option **Select call mode** to **Use 28–bit PC–region calls** (default) or to **Use 32–bit absolute calls**.

Command line syntax

–m{**f**|**n**}
––call={**far**|**near**}

Description

To address the memory of the TSK3000, you can use two different call modes:

far 32–bit absolute calls. Thought you can address the full range of memory, the address is first loaded into a register after which the call is executed.

near 28–bit PC–region call. The PC–region call is directly coded into the JAL instruction. This way of calling results in higher execution speed. However, not the full range of memory can be addressed with near calls.

If you compile your C source with near calls but the called address cannot be reached with a near call, the *linker* will generate an error.

It is recommended to use near addressing mode unless your application needs calls to addresses that fall outside a 256 MB region.

Related information

–

Compiler: ––misrac

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **C Compiler** entry and select **MISRA C**.
3. Select a MISRA C Standard.

   If you select Custom Misra C configuration:

4. In the left pane, expand the **MISRA C** entry and select **MISRA C Rules**.
5. Enable or disable the individual rules.

Command line syntax

––misrac={**all**|*number*[–*number*]**,**... }

Description

With this option you specify to the compiler which MISRA C rules must be checked. With the option **––misrac=all** the compiler checks for all supported MISRA C rules.

Example

```
c3000 --misrac=9-13 test.c
```

The compiler generates an error for each MISRA C rule 9, 10, 11, 12 or 13 violation in file `test.c`.

Related information

Compiler option **––misrac–advisory–warnings**
Compiler option **––misrac–required–warnings**

Linker option **––misra–c–report**

Compiler: --misrac-advisory-warnings / --misrac-required-warnings

Menu entry

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

   The Project Options dialog box appears.

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

3. Enable one or both options **Turn advisory rule violation into warning** and **Turn required rule violation into warning.**

Command line syntax

--misrac-advisory-warnings

--misrac-required-warnings

Description

Normally, if an advisory rule or required rule is violated, the compiler generates an error. As a consequence, no output file is generated. With this option, the compiler generates a warning instead of an error.

Related information

Compiler option **--misrac**

Linker option **--misra-c-report**

Compiler: –n (––stdout)

Menu entry

Command line only.

Command line syntax

–n
––stdout

Description

With this option you tell the compiler to send the output to stdout (usually your screen). No files are created. This option is for example useful to quickly inspect the output or to redirect the output to other tools.

Related information

–

Compiler: –O (––optimize)

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **C Compiler** entry and select **Optimization**.
3. Select an optimization level in the **Optimization level** box.
4. If you select **Custom Optimization**, enable or disable the optimizations you want.
5. In addition, in the **Size/speed trade–off** field, select a level between **fully optimize for size** or **fully optimize for speed**.

Command line syntax

–O[*flags*]
––optimize[=*flags*]

Use the following options for predefined sets of flags:

–O0 (**––optimize=0**)	**No optimization** Alias for: **–OABCEFGIKLOPRSUY**
–O1 (**––optimize=1**)	**Few optimizations** (suitable for debugging) Alias for: **–OabcefgIKLOPRSUy**
–O2 (**––optimize=2**)	**Medium optimization** (default) Alias for: **–OabcefgIkloprsUy**
–O3 (**––optimize=3**)	**Full optimization** Alias for: **–Oabcefgikloprsuy**

You can enable the following individual optimizations:

a/**A**	(+/–**coalesce**)	**Coalescer** (remove unnecessary moves)
b/**B**	(+/–**ipro**)	**Interprocedural Register Optimization**
c/**C**	(+/–**cse**)	**Common subexpression elimination (CSE)**
e/**E**	(+/–**expression**)	**Expression simplification**
f/**F**	(+/–**flow**)	**Control flow simplification** (optimization and code reordering)
g/**G**	(+/–**glo**)	**Generic assembly code optimizations**
i/**I**	(+/–**inline**)	**Function inlining**
k/**K**	(+/–**schedule**)	**Instruction scheduler**
l/**L**	(+/–**loop**)	**Loop transformations**
o/**O**	(+/–**forward**)	**Forward store**
p/**P**	(+/–**propagate**)	**Constant propagation**
r/**R**	(+/–**compact**)	**Code compaction (reverse inlining)**
s/**S**	(+/–**subscript**)	**Subscript strength reduction**
u/**U**	(+/–**unroll**)	**Unroll small loops**
y/**Y**	(+/–**peephole**)	**Peephole optimizations**

For an extensive description of these optimizations, please refer to section 4.3, *Compiler Optimizations* in chapter *Using the Compiler* of the user's manual.

Description

The TASKING C compilers offer four optimization levels and a custom level, at each level a specific set of optimizations is enabled.

- **Level 0** (**–O0**): No optimizations are performed. The compiler tries to achieve a 1–to–1 resemblance between source code and produced code. Expressions are evaluated in the order written in the source code, associative and commutative properties are not used.
- **Level 1** (**–O1**): Enables optimizations that do not affect the debug–ability of the source code. Use this level when you are developing/debugging new source code.
- **Level 2** (**–O2**): Enables more aggressive optimizations to reduce the memory footprint and/or execution time. The debugger can handle this code but the relation between source code and generated instructions may be hard to understand. Use this level for those modules that are already debugged. This is the default optimization level.
- **Level 3**: (**–O3**): Enables aggressive global optimization techniques. The relation between source code and generated instructions can be very hard to understand. The debugger does not crash, will not provide misleading information, but does not fully understand what is going on. Use this level when your program does not fit in the memory provided by your system anymore, or when your program/hardware has become too slow to meet your real–time requirements.
- **Custom level** (**–O***x*/*X*): you can enable/disable specific optimizations.

With these options you can control the level of optimization. The default optimization level is **Medium optimization** (option **–O2** or **–O** or **–OabcefgIkloprsUy**).

You can overrule these settings in your C source file with the pragma pair `#pragma optimize` *`flag`* and `#pragma endoptimize`.

In addition to the command line option **–O**, you can specify the option **–t**. With this option you specify whether the used optimizations should optimize for more speed (regardless of code size) or for smaller code size (regardless of speed).

Example

The following invocations are equivalent and result all in the default medium optimization set:

```
c3000 test.c

c3000 -O2 test.c
c3000 --optimize=2 test.c

c3000 -O test.c
c3000 --optimize test.c

c3000 -OabcefgIkloprsuy test.c
c3000 --optimize=+coalesce,+ipro,+cse,+expression,+flow,+glo,
  -inline,+schedule,+loop,+forward,+propagate,+compact,+subscript,
  +unroll,+peephole test.c
```

Related information

Section 4.3, *Compiler Optimizations*, in chapter *Using the Compiler* of the user's manual.

Compiler option **–t** (Trade off between speed (**–t0**) and size (**–t4**))

Compiler: –o (––output)

Menu entry

Altium Designer names the output file always after the C source file.

Command line syntax

–o *file*
––output=*file*

Description

With this option you can specify another filename for the output file of the compiler. Without this option the basename of the C source file is used with extension `.src`.

Example

To create the file `output.src` instead of `test.src`, enter:

```
c3000 –o output.src test.c
c3000 --output=output.src test.c
```

Related information

–

Compiler: –p (––profile)

Menu entry

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

   The Project Options dialog box appears.

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

3. Enable the option **Generate profiling information**.

4. Enable one or more of the following suboptions to select which profiles should be obtained:
   - **Block counters** (not in combination with with ***Call graph*** or ***Function timers***)
   - **Call graph**
   - **Function counters**
   - **Function timers**

Note that the more detailled information you request, the larger the overhead in terms of execution time, code size and heap space needed. The option **Generate Debug information** (**–g** or **––debug**) does not affect profiling, execution time or code size.

Command line syntax

–p[*flags*]
––profile[=*flags*]

Use the following option for a predefined set of flags:

–pg	(**––profile=g**)	profiling with call graph and function timers Alias for: **–pBcFt**

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

b/B	(+/–**block**)	**block counters**
c/C	(+/–**callgraph**)	**call graph**
f/F	(+/–**function**)	**function counters**
t/T	(+/–**time**)	**function timers**

Description

Profiling is the process of collecting statistical data about a running application. With these data you can analyze which functions are called, how often they are called and what their execution time is.

Several methods of profiling exists. One method is *code instrumentation* which adds code to your application that takes care of the profiling process when the application is executed.

For an extensive description of profiling refer to Chapter 5, *Profiling* in the user's manual.

With this option, the compiler adds the extra code to your application that takes care of the profiling process. You can obtain the following profiling data (see flags above):

Block counters (not in combination with Call graph or Time)

This will instrument the code to perform basic block counting. As the program runs, it counts the number of executions of each branch in `an if` statement is executed, each iteration of a `for` loop, and so on. Note that though you can combine Block counters with Function counters, this has no effect because Function counters is only a subset of Block counters.

Call graph (not in combination with Block counters)

This will instrument the code to reconstruct the run–time call graph. As the program runs it associates the caller with the gathered profiling data.

Function counters

This will instrument the code to perform function call counting. This is a subset of the basic Block counters.

Time (not in combination with Block counters)

This will instrument the code to measure the time spent in a function. This includes the time spend in all sub functions (callees).

If you use the profiling option, you must link the corresponding libraries too! Refer to Section 7.4, *Linking with Libraries* in Chapter *Linker* of the user's manual, for an overview of the (profiling) libraries. When you use Altium Designer, automatically the correct libraries are linked.

Example

To generate block count information for the module `test.c` during execution, compile as follows:

```
c3000 –pb test.c
c3000 ––profile=+block test.c
```

In this case you must link the library `pb3000.lib`.

Related information

Chapter 5, *Profiling* in the user's manual.

Compiler: –r (––runtime)

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **C Compiler** entry and select **Miscellaneous**.
3. Add the option **––runtime** to the **Additional C compiler options** field.

Command line syntax

–r[*flags*]
––runtime[=*flags*]

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

b/**B**	(+/–**bounds**)	bounds checking
c/**C**	(+/–**case**)	report unhandled case in a switch
m/**M**	(+/–**malloc**)	malloc consistency checks

Description

This option controls a number of run–time checks to detect errors during program execution. Some of these checks require additional code to be inserted in the original application code, and may therefore slow down the program execution. The following checks are available:

bounds

Every pointer update and dereference will be checked to detect out–of–bounds accesses, null pointers and uninitialized automatic pointer variables. This check will increase the code size and slow down the program considerably. In addition, some heap memory is allocated to store the bounds information. You may enable bounds checking for individual modules or even parts of modules only (see #pragma runtime).

case

Report an unhandled case value in a switch without a default part. This check will add one function call to every switch without a default part, but it will have little impact on the excution speed.

malloc

This option enables the use of wrappers around the functions malloc/realloc/free that will check for common dynamic memory allocation errors like:

- buffer overflow
- write to freed memory
- multiple calls to free
- passing invalid pointer to free

Enabling this check will extract some additional code fromc the library, but it will not enlarge your application code. The dynamic memory usage will increase by a couple of bytes per allocation.

Related information

–

Compiler: –s (––source)

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **C Compiler** entry and select **Miscellaneous**.
3. Enable the option **Merge C source code with assembly in output file (.src)**.

Command line syntax

–s
––source

Description

With this option you tell the compiler to merge C source code with generated assembly code in the output file. The C source lines are included as comments.

Related information

–

Compiler: --sdata

Menu entry

Command line only.

Command line syntax

--sdata=*size* (Default: 4 bytes)

Description

Without this option, all data objects of 4 bytes and smaller are placed into the small data sections and small bss sections.

With this option you tell the compiler to place all global and static data objects smaller than the specified *size* (bytes) into the small data section (sdata) or small bss section (sbss). This results in smaller and faster code. In total, 64kB is available for this kind of addressing.

You can still overrule this option with the keywords `__sdata` and `__no_sdata` for individiual data objects in your source.

If you use this option, you must use this option with the same value for all modules in your application.

Example

To put all global and static data objects with a size of 8 bytes or smaller into the sdata section:

```
c3000 --sdata=8 test.c
```

Related information

Section 2.3, *Memory Qualifiers* of the user's manual.

Compiler: −−signed−bitfields

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **C Compiler** entry and select **Language**.
3. Enable the option **Treat 'int' bit−fields as signed**.

Command line syntax

−−signed−bitfields

Description

For bit−fields it depends on the implementation whether a plain `int` is treated as `signed int` or `unsigned int`. By default an `int` bit−field is treated as `unsigned int`. This offers the best performance. With this option you tell the compiler to treat `int` bit−fields as `signed int`. In this case, you can still add the keyword `unsigned` to treat a particular `int` bit−field as `unsigned`.

Related information

−

Compiler: ––static

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **C Compiler** entry and select **Miscellaneous**.
3. Add the option **––static** to the **Additional C compiler options** field.

Command line syntax

––static

Description

With this option, the compiler treats external definitions at file scope (except for `main`) as if they were declared `static`. As a result, unused functions will be eliminated, and the alias checking algorithm assumes that objects with static storage cannot be referenced from functions outside the current module.

On the command line this option only makes sense when you specify all modules of an application on the command line.

Example

```
c3000 --static module1.c module2.c module3.c ...
```

Related information

–

Compiler: –t (––tradeoff)

Menu entry

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

   The Project Options dialog box appears.

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

3. In the **Size/speed trade–off** field, select a level between **fully optimize for size** or **fully optimize for speed**.

Command line syntax

–t{**0**|**1**|**2**|**3**|**4**}
––tradeoff={**0**|**1**|**2**|**3**|**4**}

Description

If the compiler uses certain optimizations (option **–O**), you can use this option to specify whether the used optimizations should optimize for more speed (regardless of code size) or for smaller code size (regardless of speed).

Default the compiler optimizes for more speed (**–t0**).

If you have not used the option **–O**, the compiler uses default medium optimization, so you can still specify the option **–t**.

Related information

Compiler option **–O** (Specify optimization level)

Compiler: –U (––undefine)

Menu entry

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

   The Project Options dialog box appears.

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

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

Command line syntax

–U*macro_name*
––undefine=*macro_name*

Description

With this option you can undefine an earlier defined macro as with `#undef`.

This option is for example useful to undefine predefined macros.

However, the following predefined ISO C standard macros cannot be undefined:

__FILE__	current source filename
__LINE__	current source line number (int type)
__TIME__	hh:mm:ss
__DATE__	mmm dd yyyy
__STDC__	level of ANSI standard

Example

To undefine the predefined macro __TASKING__:

```
c3000 -U__TASKING__ test.c
c3000 --undefine=__TASKING__ test.c
```

Related information

Compiler option **–D** (Define preprocessor macro)

Compiler: –u (––uchar)

Menu entry

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

   The Project Options dialog box appears.

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

3. Enable the option **Treat 'char' variables as unsigned**.

Command line syntax

–u
––uchar

Description

By default `char` is the same as specifying `signed char`. With this option `char` is the same as `unsigned char`.

Related information

–

Compiler: ––use–hardware

Menu entry

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

   The Project Options dialog box appears.

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

3. Enable one or more of the options:

   - **Multiply/Divide unit present**

Command line syntax

––use–hardware=*flag*,...

You can set the following flags:

d/**D** (+/–**divide**)	**Divide instructions**
m/**M** (+/–**multiply**)	**Multiply instructions**

Default**: dm**

Description

With this option you tell the compiler that the TSK3000 target has a hardware multiply/divide unit. This way the compiler can use the optional divide and multiply instructions.

Related information

–

Compiler: –V (––version)

Menu entry

Command line only.

Command line syntax

–V
––version

Description

Displays version information of the compiler. The compiler ignores all other options or input files.

Related information

–

Compiler: –w (––no–warnings)

Menu entry

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

   The Project Options dialog box appears.

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

3. In the Warnings field, select one of the following options:

   - **Report all warnings**
   - **Suppress all warnings**
   - **Suppress specific warningsn**

   *If you select **Suppress specific warnings**:*

4. Enter the numbers, separated by commas, of the warnings you want to suppress.

Command line syntax

–w[*nr*]
––no–warnings[=*nr*]

Description

With this option you can suppress 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 **–w** multiple times.

Example

To suppress warnings 135 and 136, enter **135, 136** in the **Specific warnings to suppress** field, or enter the following on the command line:

```
c3000 test.c –w135 –w136
c3000 test.c ––no–warnings=135 ––no–warnings=136
```

Related information

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

Compiler: −−warnings−as−errors

Menu entry

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

   The Project Options dialog box appears.

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

3. Enable the option **Treat warnings as errors**.

Command line syntax

−−warnings−as−errors

Description

If the compiler encounters an error, it stops compiling. With this option you tell the compiler to treat warnings as errors. As a consequence, the compiler now also stops after encountering a warning.

Related information

Compiler option **−w** (Suppress some or all warnings)

4.2 Assembler Options

Altium Designer uses a makefile to build your entire project. This means that in Altium Designer you cannot run the assembler separately. If you want assembly results, you must compile a single C source file from within Altium Designer, the file is then also assembled. However, you can set options specific for the assembler.

Options in Altium Designer versus options on the command line

Most command line options have an equivalent option in Altium Designer but some options are only available on the command line (for example in a Windows Command Prompt). If there is no equivalent option in Altium Designer, you can specify a command line option in Altium Designer as follows:

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

   The Project Options dialog box appears.

2. Expand the **Assembler** entry and select **Miscellaneous**.

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

Invocation syntax on the command line (Windows Command Prompt)

To call the assembler from the command line, use the following syntax:

```
as3000 [ [option]... [file]... ]...
```

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

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 double minus (––) characters. You can abbreviate long option names as long as the name is unique. You can mix short and long option names on the command line.

Options can have flags or sub–options. 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:

```
as3000 -Ogs test.src

as3000 --optimize=+generics,+instr-size test.src
```

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

Assembler: –? (––help)

Menu entry

Command line only.

Command line syntax

–?
––help[=**options**]

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:

```
as3000 -?
as3000 --help
as3000
```

To see a detailed description of the available options, enter:

```
as3000 --help=options
```

Assembler: –c (––case–insensitive)

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Assembler** entry and select **Miscellaneous**.

3. Disable the option **Assemble case sensitive**.

Command line syntax

–c
––case–insensitive

Description

With this option you tell the assembler not to distinguish between upper and lower case characters. By default the assembler considers upper and lower case characters as different characters.

Disabling the option **Assemble case sensitive** in Altium Designer is the same as specifying the option **–c** on the command line.

Assembly source files that are generated by the compiler must always be assembled case sensitive. When you are writing your own assembly code, you may want to specify the case insensitive mode.

Example

When assembling case insensitive, the label `LabelName` is the same label as `labelname`.

Related information

–

Assembler: −−check

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Assembler** entry and select **Miscellaneous**.
3. Add the option **−−check** to the **Additional assembler options** field.

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.

Related information

Compiler option **−−check** (Check syntax)

Assembler: –D (––define)

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Assembler** entry and select **Preprocessing**.
3. Click on **User macro**, click on the down arrow in the right pane to expand macro input.
4. Click on an empty **Macro** field and enter a macro name. (Then click outside the cell to confirm)
5. Optionally, click in the **Value** field and enter a definition. (Then click outside the cell to confirm)

Command line syntax

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

Description

With this option you can define a macro and specify it to the assembler preprocessor. If you only specify a macro name (no macro definition), the macro expands as ’1’.

You can specify as many macros as you like. On the command line you can use the option **–D** multiple times. If the command line exceeds the limit of the operating system, you can define the macros in an *option file* which you then must specify to the assembler with the option **–f***file*.

Defining macros with this option (instead of in the assembly source) is, for example, useful in combination with conditional assembly as shown in the example below.

This option has the same effect as defining symbols via the `.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:

Macro	Value
`DEMO`	1 (or empty)

```
   as3000 -DDEMO test.src
   as3000 -DDEMO=1 test.src

   as3000 --define=DEMO test.src
   as3000 --define=DEMO=1 test.src
```

Note that all four invocations have the same effect.

Related information

Assembler option **–f** (Read options from file)

Assembler: --diag

Menu entry

1. From the **View** menu, select **Workspace Panels** » **System** » **Messages**.

   The Messages panel appears.

2. In the **Messages** panel, right-click on the message you want more information on.

   A popup menu appears.

3. Select **More Info**.

   A Message Info box appears with additional information.

Command line syntax

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

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to `stdout` (normally your screen) and in the format you specify. You can specify the following formats: **html**, **rtf** or **text** (default). To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas.

With this option the assembler does not assemble any files.

Example

To display an explanation of message number 241, enter:

```
as3000 --diag=241
```

This results in the following message and explanation:

```
W241: additional input files will be ignored

The assembler supports only a single input file. All other input files
are ignored.
```

To write an explanation of all errors and warnings in HTML format to file `aserrors.html`, use redirection and enter:

```
as3000 --diag=html:all > aserrors.html
```

Related information

–

Assembler: −−emit−locals

Menu entry

Command line only.

Command line syntax

−−emit−locals

Description

With this option the assembler also emits local symbols to the object file. Normally, only global symbols are emitted. Having local symbols in the object file can be useful for debugging.

Related information

−

Assembler: ––error–file

Menu entry

Command line only.

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.err` instead of `stderr`, enter:

```
as3000 --error-file=errors.err test.src
```

Related information

–

Assembler: –f (––option–file)

Menu entry

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

   The Project Options dialog box appears.
2. Expand the **Assembler** entry and select **Miscellaneous**.
3. Add the option **–f** to the **Additional assembler options** field.

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

Command line syntax

–f *file*
––option–file=*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 **–f** multiple times.

Format of an option file

- Multiple arguments on one line in the option file are allowed.
- To include whitespace in an argument, surround the argument with single or double quotes.
- If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

```
"This has a single quote ' embedded"

'This has a double quote " embedded'

'This has a double quote " and a single quote '"' embedded"
```

- When a text line reaches its length limit, use a 'to continue the line. Whitespace between quotes is preserved.

```
"This is a continuation \
line"

    -> "This is a continuation line"
```

- It is possible to nest command line files up to 25 levels.

Example

Suppose the file `myoptions` contains the following lines:

```
-gaL
test.src
```

Specify the option file to the assembler:

```
as3000 -f myoptions
as3000 --option-file=myoptions
```

This is equivalent to the following command line:

```
as3000 -gaL test.src
```

Related information

–

Assembler: −−gp−relative

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Assembler** entry and select **Miscellaneous**.

3. Enable the option **Automatically generate GP−relative offsets**.

Command line syntax

−−gp−relative

Description

When this option is enabled the assembler automatically emits a GP−relative relocation for symbolic offsets in load and store instructions if the base register is the GP−register ($28).

When this option is disabled (default) you must use the built−in assembly function `@GPREL()` on the symbolic offset in order to force the assembler to emit a GP−relative relocation. Assembly code generated by the compiler always uses the `@GPREL()` function.

Related information

Assembly function `@GPREL()`

Assembler: –g (––debug–info)

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Assembler** entry and select **Debug Information**.

3. Select which debug information to include: **Automatic HLL or assembly level debug information**, **Custom debug information** or **No debug information**.

   *If you select **Custom debug information**:*

4. Select which Custom debug information to include: **Assembler source line information**, **Pass HLL debug information**, or **None**.

5. Enable or disable the option **Assembler local symbols information**.

Command line syntax

–g[*flag*]
––debug–info[=*flag*]

You can set the following flags:

a/**A**	(+/–**asm**)	Assembly source line information
h/**H**	(+/–**hll**)	Pass high level language debug information (HLL)
l/**L**	(+/–**local**)	Assembler local symbols debug information (default)
s/**S**	(+/–**smart**)	Smart debug information (default)

If you do not use this option or if you specify **–g** without any flags, the default is **–gsl**.

You cannot specify **–gah**. Either the assembler generates assembly source line information, or it passes HLL debug information.

When you specify **–gs**, the assembler selects which flags to use. If high level language information is available in the source file, the assembler passes this information (same as **–gAhL**).
If not, the assembler generates assembly source line information (same as **–gaHl**).

With **–gAHLS** the assembler does not generate any debug information.

Description

With this option you tell the assembler which kind of debug information to emit in the object file.

Related information

–

Assembler: –H (––include–file)

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Assembler** entry and select **Preprocessing**.
3. Enter the name of the file in the **Include this file before source** field or click **...** and select a file.

Command line syntax

–H*file*,...
––include–file=*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

```
as3000 -Hmyinc.inc test1.src
as3000 --option-file=myoptions
```

The file `myinc.inc` is included at the beginning of `test1.src` before it is assembled.

Related information

Assembler option **–I** (Include files path)

Section 6.5, *How the Assembler Searches Include Files*, in chapter *Using the Assembler* of the user's manual.

Assembler: –I (––include–directory)

Menu entry

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

   The Project Options dialog box appears.

2. Select **Build Options**.
3. Add a pathname in the **Include Files Path** field.

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

Command line syntax

–I*path,...*
––include–directory=*path,...*

Description

With this option you can specify the path where your include files are located. A relative path will be relative to the current directory.

The order in which the 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 `AS3000INC` when the product was installed.
4. The default `include` directory relative to the installation directory.

Example

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

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

You can call the assembler as follows:

```
as3000 -Ic:\proj\include test.src
as3000 --include-directory=c:\proj\include test.src
```

First the assembler looks in the directory where `test.src` is located for the file `myinc.inc`. If it does not find the file, it looks in the directory `c:\proj\include` for the file `myinc.inc` (this option). 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 **–H** (**––include–file**) (Include file before source)

Assembler: –i (––symbol–scope)

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Assembler** entry and select **Miscellaneous**.
3. Select the default label mode: **Local** or **Global**.

Command line syntax

```
–i{g|l}                                   (Default: –il)
––symbol–scope={global|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: –k (––keep–output–files)

Menu entry

Altium Designer *always* removes the object file when errors occur during assembling.

Command line syntax

–k
––keep–output–files

Description

If an error occurs during assembling, the resulting object file (`.obj`) may be incomplete or incorrect. With this option you keep the generated object file when an error occurs.

By default the assembler removes the generated object file when an error occurs. This is useful when you use the make utility. If the erroneous files are not removed, the make utility may process corrupt files on a subsequent invocation.

Use this option when you still want to use the generated object. For example when you know that a particular error does not result in a corrupt object file.

Related information

–

Assembler: –L (––list–format)

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Assembler** entry and select **List File**.
3. Enable **Generate list file**.
4. In the **List file format** section, enable or disable the types of information to be included.

Command line syntax

–L*flags*
––list–format=*flags*

You can set the following flags:

0	Same as **–LDEGIMNPQRSVWXYZ** (all options disabled)
1	Same as **–Ldegimnpqrsvwxyz** (all options enabled)

d/D	(+/–**section**)	**Section directives (.SECTION)**
e/E	(+/–**symbol**)	**Symbol definition directives**
g/G	(+/–**generic–expansion**)	**Generic instruction expansion**
i/I	(+/–**generic**)	**Generic instructions**
m/M	(+/–**macro**)	**Macro/dup definitions (e.g. .MACRO)**
n/N	(+/–**empty–line**)	**Empty source lines (newline)**
p/P	(+/–**conditional**)	**Conditional assembly (.IF, .ELSE, .ENDIF)**
q/Q	(+/–**equate**)	**Assembler .EQU and .SET directives**
r/R	(+/–**relocations**)	**Relocation characters ('r')**
s/S	(+/–**hll**)	**HLL symbolic debug information (.SYMB)**
v/V	(+/–**equate–values**)	**Assembler .EQU and .SET values**
w/W	(+/–**wrap–lines**)	**Wrapped source lines**
x/X	(+/–**macro–expansion**)	**Macro expansions**
y/Y	(+/–**cycle–count**)	**Cycle counts**
z/Z	**(+/–macro–expansion)**	**Define expansions**

Default: **–LdEGiMnPqrsVWXyZ**

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 **–l (––list–file)**.

Related information

Assembler option **–l** (Generate list file)
Assembler option **–tl** (Display section information in list file)

Assembler: –l (––list–file)

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Assembler** entry and select **List File**.
3. Enable **Generate list file**.
4. In the **List file format** section, enable or disable the types of information to be included.

Command line syntax

–l[*file*]
––list–file[=*file*]

Description

With this option you tell the assembler to generate a list file. A list file shows the generated object code and the relative addresses. Note that the assembler generates a relocatable object file with relative addresses.

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

On the command line you can use the option **–L** (**––list–format**) to specify which types of information should be included in the list file.

Assembler: –m (––preprocessor–type)

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Assembler** entry and select **Miscellaneous**.

3. Add the option **–m** to the **Additional assembler options** field.

Command line syntax

–m{**n**|**t**}	Default: **–mt**
––preprocessor–type={**none**|**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: ––nop–insertion

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Assembler** entry and select **Miscellaneous**.

3. Enable the option **Insert a NOP after all jumps and branches**.

Command line syntax

––nop–insertion

Description

When this option is enabled the assembler automatically fills the delay slots of jump and branch instructions with a NOP instruction. NOP insertion can be done with higher granularity by using the `.nopinsertion` and `.nonopinsertion` directives in assembly sources.

Related information

Assembler directive `.nopinsertion`

Assembler: –O (––optimize)

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Assembler** entry and select **Optimization**.

3. Enable or disable the optimization options:

   - **Generic instructions**
   - **Jumpchains**
   - **Instruction size**

Command line syntax

–O*flags*
––optimize=*flags*

You can set the following flags:

g/**G**	(+/–**generics**)	Allow **generic instructions**
j/**J**	(+/–**jumpchains**)	**Jump chains**
s/**S**	(+/–**instr–size**)	Optimize **instruction size**

Default**: –OgJs**

Description

Allow generic instructions

If you use generic instructions in your assembly source, the assembler can optimize them by replacing it with the fastest or shortest possible variant of that instruction. By default this option is enabled. If you turn off this optimization, the assembler generates an error on generic instructions. Be aware that the compiler also generates generic instructions!

Jump chains

With this optimization, the assembler replaces chained jumps by a single jump instruction. For example, a jump from *a* to *b* immediately followed by a jump from *b* to *c*, is replaced by a jump from *a* to *c*.

Optimize instruction size

With this optimization the assembler tries to find the shortest possible operand encoding for instructions.

Related information

Section 6.3, *Assembler Optimizations* in chapter *Using the Assembler* of the user's manual.

Assembler: –o (––output)

Menu entry

Altium Designer names the output file always after the source file.

Command line syntax

–o *file*
––output=*file*

Description

With this option you can specify another filename for the output file of the assembler. Without this option, the basename of the assembly source file is used with extension `.obj`.

Example

To create the file `relobj.obj` instead of `asm.obj`, enter:

```
as3000 -o relobj.obj asm.src
as3000 --output=relobj.obj asm.src
```

Related information

–

Assembler: –t (––section–info)

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Assembler** entry and select **List File**.
3. Enable **Generate list file**.
4. Enable the option **Display section information**.

Command line syntax

–t*flags*
––section–info=*flags*

You can set the following flags:

c/**C** (+/–**console**)	Display section information on `stdout`.
l/**L** (+/–**list**)	Write section information to the list file.

Description

With this option you tell the assembler to display section information. For each section its memory space, size, total cycle counts and name is listed on stdout and/or in the list file.

The cycle count consists of two parts: the total accumulated count for the section and the total accumulated count for all repeated instructions. In the case of nested loops it is possible that the total supersedes the section total.

With **–tl**, the assembler writes the section information to the list file. You must specify this option in combination with the option **–l** (generate list file).

Example

```
as3000 –l –tcl test.src
as3000 –l ––section–info=+console,+list test.src
```

The assembler generates a list file and writes the section information to this file. The section information is also displayed on stdout.

Related information

Assembler option **–l** (generate list file)

Assembler: ––use–hardware

Menu entry

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

   The Project Options dialog box appears.

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

3. Enable one or more of the options:

   - **Multiply/Divide unit present**

Command line syntax

––use–hardware=*flag*,...

You can set the following flags:

d/**D** (+/–**divide**)	**Divide instructions**
m/**M** (+/–**multiply**)	**Multiply instructions**

Default**: dm**

Description

With this option you tell the assembler that the TSK3000 target has a hardware multiply/divide unit. This way the assembler can use the optional divide and multiply instructions.

Related information

–

Assembler: –V (––version)

Menu entry

Command line only.

Command line syntax

–V
––version

Description

Displays version information of the assembler. The assembler ignores all other options or input files.

Related information

–

Assembler: –v (––verbose)

Menu entry

Command line only.

Command line syntax

–v
––verbose

Description

With this option you put the assembler in verbose mode. The assembler prints the filenames and the assembly passes while it processes the files so you can monitor the current status of the assembler.

Related information

–

Assembler: –w (––no–warnings)

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Assembler** entry and select **Diagnostics**.
3. Enable one of the options:
   - **Report all warnings**
   - **Suppress all warnings**
   - **Suppress specific warnings**

   *If you select **Suppress specific warnings**:*

4. Enter the numbers, separated by commas, of the warnings you want to suppress.

Command line syntax

–w[*nr*]
––no–warnings[=*nr*]

Description

With this option you can suppresses all warning messages or specific warning messages.

- If you do not specify this option, all warnings are reported.
- If you specify this option but without numbers, all warnings are suppressed.
- If you specify this option with a number, only the specified warning is suppressed.
  You can specify the option **–w** multiple times.

Example

To suppress warnings 135 and 136, enter **135, 136** in the **Specific warnings to suppress** field, or enter the following on the command line:

```
as3000 test.src -w135 -w136
as3000 test.src --no-warnings=135 --no-warnings=136
```

Related information

Assembler option **––warnings–as–errors** (Treat warnings as errors)

Assembler: −−warnings−as−errors

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Assembler** entry and select **Diagnostics**.

3. Enable the option **Treat warnings as errors**.

Command line syntax

−−warnings−as−errors

Description

If the assembler encounters an error, it stops assembling. With this option you tell the assembler to treat warnings as errors. As a consequence, the assembler now also stops after encountering a warning.

Related information

Assembler option **−w** (Suppress some or all warnings)

4.3 Linker Options

Altium Designer uses a *makefile* to build your entire project. This means that you cannot run the linker separately. However, you can set options specific for the linker.

Options in Altium Designer versus options on the command line

Most command line options have an equivalent option in Altium Designer but some options are only available on the command line (for example in a Windows Command Prompt). If there is no equivalent option in Altium Designer, you can specify a command line option in Altium Designer as follows:

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

   The Project Options dialog box appears.

2. Expand the **Linker** entry and select **Miscellaneous**.
3. Enter one or more command line options in the **Additional Linker options** field.

Invocation syntax on the command line (Windows Command Prompt)

The invocation syntax on the command line is:

```
tlk [ [option]... [file]... ]...
```

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

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 double minus (––) characters. You can abbreviate long option names as long as the name is unique. You can mix short and long option names on the command line.

Options can have flags or sub–options. 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:

```
tlk -mfk test.obj

tlk --map-file-format=+files,+link test.obj
```

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

Linker: –? (––help)

Menu entry

–

Command line syntax

–?
––help[=**options**]

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:

```
tlk -?
tlk --help
tlk
```

To see a detailed description of the available options, enter:

```
tlk --help=options
```

Linker: –c (––chip–output)

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Linker** entry and select **Output Format**.
3. Enable the options **Intel HEX records** and/or **Motorola S–records**.

Command line syntax

–c[*basename*]**:***format*[**:***addr_size*],...
––chip–output=[*basename*]**:***format*[**:***addr_size*],...

You can specify the following formats:

IHEX	Intel Hex
SREC	Motorola S–records

The *addr_size* specifies the size of the addresses in bytes (record length). For Intel Hex you can use the values **1**, **2** or **4** bytes (default). For Motorola–S you can specify: **2** (S1 records), **3** (S2 records) or **4** bytes (S3 records, default). In Altium Designer you cannot specify the address size because Altium Designer always uses the default values.

Description

With this option you specify the Intel Hex or Motorola S–record output format for loading into a PROM–programmer. The linker generates a file for each ROM memory defined in the LSL file, where sections are located:

```
memory memname
{  type=rom;  }
```

The name of the file is the name of the Altium Designer 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 either IEEE–695 or 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:

```
tlk -cmyfile:IHEX test1.obj

tlk --chip-output=myfile:IHEX test1.obj
```

In this case, this generates the file `myfile_`*`memname`*`.hex`

Related information

Linker option **-o** (Output file)

Section 6.2, *Motorola S-Record Format*,
Section 6.3, *Intel Hex Record Format*, in Chapter *Object File Formats*.

Linker: ––case–insensitive

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Linker** entry and select **Miscellaneous**.
3. *Disable* the option **Link case sensitive**.

Command line syntax

––case–insensitive

Description

With this option you tell the linker not to distinguish between upper and lower case characters in symbols. By default the linker considers upper and lower case characters as different characters.

Disabling the option **Link case sensitive** in Altium Designer is the same as specifying the option **––case–insensitive** on the command line.

Assembly source files that are generated by the compiler must *always* be assembled and thus linked case sensitive. When you have written your own assembly code and specified to assemble it case insensitive, you must also link the `.obj` file case insensitive.

Related information

–

Linker: –D (––define)

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Linker** entry and select **Miscellaneous**.
3. Add the option **–D** to the **Additional linker options** field.

Command line syntax

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

Description

With this option you can define a macro and specify it to the linker 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 **–D** multiple times. If the command line exceeds the limit of the operating system, you can define the macros in an *option file* which you then must specify to the linker with the option –**f***file*.

The definition can be tested by the preprocessor with `#if`, `#ifdef` and `#ifndef`, for conditional locating.

Example

To define the stack size and start address which are used in the linker script file `3000.lsl`, enter:

```
tlk test.obj -otest.abs -d3000.lsl -D__STACK=32k
    -D__START=0x00000000
```

or using the long option names:

```
tlk -otest.abs -lsl-file=3000.lsl --define=__STACK=32k
    --define=__START=0x00000000
```

Related information

Linker option **–f** (Read options from file)

Linker: –d (––lsl–file)

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Linker** entry and select **Miscellaneous**.
3. Enable the option **Use project specific LSL file**.
4. In the **LSL file** field, type a name or click **...** and select an LSL file.

Command line syntax

–d*file*
––lsl–file=*file*

Description

A linker script file contains vital information about the core for the locating phase of the linker. A linker script file is coded in LSL and contains the following types of information:

- the architecture 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 `3000.lsl` or the name of a manually written linker script file. You can use this option multiple times. The linker processes the LSL files in the order in which they appear on the command line.

Related information

Linker option **––lsl–check** (Check LSL file(s) and exit)

Section 7.7, *Controlling the Linker with a Script*, in chapter *Using the Linker* of the user's manual.

Linker: --diag

Menu entry

1. From the **View** menu, select **Workspace Panels** » **System** » **Messages**.

   The Messages panel appears.

2. In the **Messages** panel, right-click on the message you want more information on.

   A popup menu appears.

3. Select **More Info**.

   A Message Info box appears with additional information.

Command line syntax

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

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to `stdout` (normally your screen) and in the format you specify. You can specify the following formats: **html**, **rtf** or **text** (default). To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas.

With this option the linker does not link/locate any files.

Example

To display an explanation of message number 106, enter:

```
tlk --diag=106
```

This results in the following message and explanation:

```
E106: unresolved external: <message>

The linker could not resolve all external symbols. This is an error when
the incremental linking option is disabled. The <message> indicates the
symbol that is unresolved.
```

To write an explanation of all errors and warnings in HTML format to file `lerrors.html`, enter:

```
tlk --diag=html:all > lerrors.html
```

Related information

-

Linker: –e (––extern)

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Linker** entry and select **Miscellaneous**.

3. Add the option **–e** to the **Additional linker options** field.

Command line syntax

–e *symbol*
––extern=*symbol*

Description

With this option you force the linker to consider the given symbol as an undefined reference. The linker tries to resolve this symbol, 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:

```
tlk mylib.lib
```

Nothing is linked and no output file will be produced, because there are no unresolved symbols when the linker searches through `mylib.lib`.

```
tlk -e __START mylib.lib
tlk --extern=__START mylib.lib
```

In this case the linker searches for the symbol `__START` in the library and (if found) extracts the object that contains `__START`, the startup code. If this module contains new unresolved symbols, the linker looks again in `mylib.lib`. This process repeats until no new unresolved symbols are found.

Related information

Section 7.4, *Linking with Libraries*, in chapter *Using the Linker* of the user's manual.

Linker: ––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 `lk3000.elk`.

Example

To write errors to `errors.elk` instead of `stderr`, enter:

```
tlk --error-file=errors.elk test.obj
```

Related information

–

Linker: –f (––option–file)

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Linker** entry and select **Miscellaneous**.

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

Be aware that when you specify the option **–f** in the **Additional linker options** field, the options are added to the linker options you have set in the other dialogs. Only in extraordinary cases you may want to use them in combination. Altium Designer automatically saves the options with your project.

Command line syntax

–f *file*
––option–file=*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 length of the command line would exceed the limits of the operating system, or just to store options and save typing.

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

Format of an option file

- Multiple 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:

`-Mmymap`	(generate a map file)
`test.obj`	(input file)
`-Lc:\mylibs`	(additional search path for system libraries)

Specify the option file to the linker:

```
tlk -f myoptions
tlk --option-file=myoptions
```

This is equivalent to the following command line:

```
tlk -Mmymap test.obj -Lc:\mylibs
```

Related information

–

Linker: ––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:

```
tlk --first-library-first a.lib test.obj b.lib
```

If the file `test.obj` calls a function which is both present in `a.lib` and `b.lib`, normally the function in `b.lib` would be extracted. With this option the linker first tries to extract the symbol from the first library `a.lib`.

Note that routines in `b.lib` that call other routines that are present in both `a.lib` and `b.lib` are now also resolved from `a.lib`.

Related information

Linker option **––no–rescan** (Rescan libraries to solve unresolved externals)

Linker: –I (––include–directory)

Menu entry

–

Command line syntax

–I*path,...*
––include–directory=*path,...*

Description

With this option you can specify the path where your LSL include files are located. A relative path will be relative to the current directory.

The order in which the linker searches for LSL include files is:

1. The pathname in the LSL file and the directory where the LSL 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:

```
tlk -Ic:\proj\include -dmylsl.lsl test.obj
tlk --include-directory=c:\proj\include --lsl-file=mylsl.lsl test.obj
```

First the linker looks in the directory where `mylsl.lsl` is located for the file `myinc.inc`. If it does not find the file, it looks in the directory `c:\proj\include` for the file `myinc.inc` (this option). Finally it looks in the directory `$(PRODDIR)\include.lsl`.

Related information

–

Linker: –i (––user–provided–initialization–code)

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Linker** entry and select **Miscellaneous**.

3. Add the option **––user–provided–initialization–code** to the **Additional linker options** field.

Command line syntax

–i
––user–provided–initialization–code

Description

It is possible to use your own initialization code, for example, to save ROM space. With this option you tell the linker not to generate a copy table for initialize/clear sections. Use linker labels in your source code to access the positions of the sections when located.

If the linker detects references to the TASKING initialization code, an error is emitted: it is either the TASKING initialization routine or your own, not both.

Note that the options **––no–rom–copy** and **––non–romable**, may vary independently. The 'copytable–compression' optimization is automatically disabled when you enable this option.

Related information

–

Linker: –k (––keep–output–files)

Menu entry

Altium Designer *always* removes the output files when errors occurred.

Command line syntax

–k
––keep–output–files

Description

If an error occurs during linking, the resulting output file may be incomplete or incorrect. With this option you keep the generated output files when an error occurs.

By default the linker removes the generated output 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: –L (––library–directory / ––ignore–default–library–path)

Menu entry

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

   The Project Options dialog box appears.

2. Open the **Build Options** page.

3. Add a pathname in the **Library files path** field.

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

Command line syntax

–L*dir*
––library–directory=*dir*

–**L**
––ignore–default–library–path

Description

With this option you can specify the path(s) where your system libraries, specified with the **–l** option, are located. If you want to specify multiple paths, use the option **–L** for each separate path.

The default path is `$(PRODDIR)\c3000\lib`.

If you specify only **–L** (without a pathname) or the long option **––ignore–default–library–path**, the linker will not search the default path and also not in the paths specified in the environment variable `LIBTSK3000`. So, the linker ignores steps 2 and 3 as listed below.

The priority order in which the linker searches for system libraries specified with the **–l** option is:

1. The path that is specified with the **–L** option.

2. The path that is specified in the environment variable `LIBTSK3000`.

3. The default directory `$(PRODDIR)\c3000\lib` (or a processor specific sub–directory).

Example

Suppose you call the linker as follows:

```
tlk test.obj -Lc:\mylibs -lc3000
tlk test.obj --library-directory=c:\mylibs --library=c3000
```

First the linker looks in the directory `c:\mylibs` for library `c3000.lib` (this option).

If it does not find the requested libraries, it looks in the directory that is set with the environment variable `LIBTSK3000`.

Then the linker looks in the default directory `$(PRODDIR)\c3000\lib` for libraries.

Related information

Linker option **–l** (Link system library)

Section 7.4.1, *How the linker searches libraries* in chapter *Using the Linker* of the user's manual.

Linker: –l (––library)

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Linker** entry and select **Libraries**.

3. Enable the option **Link default C libraries**.

Command line syntax

–l*name*
––library=*name*

Description

With this option you tell the linker to use system library `name.lib`, where *name* is a string. The linker first searches for system libraries in any directories specified with **–L***path*, then in the directories specified with the environment variable `LIBTSK3000`, unless you used the option **–L** without a directory.

Example

To search in the system library `c3000.lib` (C library):

```
tlk test.obj mylib.lib –lc3000
tlk test.obj mylib.lib ––library=c3000
```

The linker links the file `test.obj` and first looks in `mylib.lib` (in the current directory only), then in the system library `c3000.lib` to resolve unresolved symbols.

Related information

Linker option **–L** (Additional search path for system libraries)

Section 7.4, *Linking with Libraries*, in chapter *Using the Linker* of the user's manual.

Linker: −−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 **−cl** (Stop after linking)

Linker: ––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 **–d***file* to specify the name of the Linker Script File you want to test.

Related information

Linker option **–d** (Linker script file)
Linker option **––lsl–dump** (Dump LSL info)

Section 7.7, *Controlling the Linker with a Script*, in chapter *Using the Linker* of the user's manual.

Linker: −−lsl−dump

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Linker** entry and select **Miscellaneous**.
3. Enable the option **Dump processor and memory info from LSL file**.

Command line syntax

−−lsl−dump[=*file*]

Description

With this option you tell the linker to dump the LSL part of the map file in a separate file, independent of the option **−M** (generate map file). If you do not specify a filename, the file `tlk.ldf` is used.

Related information

Linker option **−m** (Map file formatting)

Linker: –M (––map–file)

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Linker** entry and select **Map File**.

3. Enable the option **Generate a memory map file (.map)**.

4. In the **Map file format** section, enable or disable the information you want to be included in the map file.

Command line syntax

–M[*file*]
––map–file[=*file*]

Description

With this option you tell the linker to generate a linker map file. If you do not specify a filename and you specfied the **–o** option, the linker uses the same basename as the output file with the extension `.map`. If you did not specify the **–o** option, the linker uses the file `task1.map`. Altium Designer 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 (`.obj`) to the linked object file. A locate part shows the absolute position of each section. External symbols are listed per space with their absolute address, both sorted on symbol and sorted on address.

Related information

With the option **–m** (map file formatting) you can specify which parts you want to place in the map file.

Section 5.2, *Linker Map File Format*, in Chapter *List File Formats*.

Linker: –m (––map–file–format)

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Linker** entry and select **Map File**.
3. Enable the option **Generate a map file (.map)**.
4. In the **Map file format** section, enable or disable the information you want to be included in the map file.

Command line syntax

–m*flags*
––map–file–format=*flags*

You can specify the following formats:

0	Same as **–mcfkLMoQrSU** (link information)
1	Same as **–mCfKlMoQRSU** (locate information)
2	Same as **–mcfklmoQrSu** (most information)
c/**C** (+/–**callgraph**)	Call graph information
f/**F** (+/–**files**)	Processed files information
k/**K** (+/–**link**)	Link result information
l/**L** (+/–**locate**)	Locate result information
m/**M** (+/–**memory**)	Memory usage information
o/**O** (+/–**overlay**)	Overlay information
q/**Q** (+/–**statics**)	Module local symbols
r/**R** (+/–**crossref**)	Cross references information
s/**S** (+/–**lsl**)	Processor and memory information
u/**U** (+/–**rules**)	Locate rules

Description

With this option you specify which information you want to include in the map file. Use this option in combination with the option **–M (––map–file)**.

If you do not specify this option, the linker uses the default: **–m2**.

Related information

Linker option **–M** (Generate map file)

Linker: −−misra−c−report

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **C Compiler** entry and select **MISRA C**.
3. Select a MISRA C configuration.
4. Enable the option **Produce a MISRA C report**.

Command line syntax

−−misra−c−report[=*file*]

Description

With this option you tell the linker to create a MISRA C Quality Assurance report. This report lists the various modules in the project with the respective MISRA C settings at the time of compilation. If you do not specify a filename, the file *name*`.mcr` is used.

Related information

Compiler option **−−misrac**

Linker: –N (––no–rom–copy)

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Linker** entry and select **Miscellaneous**.

3. Add the option **–N** to the **Additional linker options** field.

Command line syntax

–N
––no–rom–copy

Description

With this option the linker will not generate a ROM copy for data sections. A copy table is generated and contains entries to clear BSS section. 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: −−no−rescan

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Linker** entry and select **Libraries**.
3. *Disable* the option **Rescan libraries to solve unresolved externals**.

Command line syntax

−−no−rescan

Description

When the linker processes a library it searches for symbol definitions that are referenced by the objects and libraries processed so far. If the library contains a definition for an unresolved reference the linker extracts the object that contains the definition from the library. The linker processes object files and libraries in the order in which they appear on the command line.

When all objects and libraries are processed the linker checks if there are unresolved symbols left. If so, the default behavior of the linker is to rescan all libraries in the order given 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: −−non−romable

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Linker** entry and select **Miscellaneous**.
3. Add the option **−−non−romable** to the **Additional linker options** field.

Command line syntax

−−non−romable

Description

With this option, the linker will locate all ROM sections in RAM. A copy table is generated and is located in RAM. When the application is started, that data and BSS sections are re−initialized.

Related information

−

Linker: –O (––optimize)

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Linker** entry and select **Optimization**.
3. Select an optimization level in the **Optimization level** box.

   *If you select **Custom Optimization:***

4. Enable the optimizations you want.

Command line syntax

–O[*flags*]
––optimize[=*flags*]

Use the following options for predefined sets of flags:

–O0 (**––optimize=0**)	**No optimization** Alias for: **–OCLTXY**
–O1 (**––optimize=1**)	**Default optimization** Alias for: **–OCLtXY**
–O2 (**––optimize=2**)	All optimizations Alias for: **–Ocltxy**

You can set the following flags:

c/C	(+/–**delete–unreferenced–sections**)	Delete unreferenced sections from the output file (no effect on sources compiled with debug information)
l/L	(+/–**first–fit–decreasing**)	Use a 'first fit decreasing' algorithm to locate unrestricted sections in memory.
t/T	(+/–**copytable–compression**)	Emit smart restrictions to reduce copy table size
x/X	(+/–**delete–duplicate–code**)	Delete duplicate code sections from the output file
y/Y	(+/–**delete–duplicate–data**)	Delete duplicate constant data from the output file

Description

With this option you can control the level of optimization the linker performs. If you do not use this option, **–OCLtXY** (**–O1**) is the default.

Related information

Section 7.2.3, *Linker Optimizations*, in chapter *Using the Linker* of the user's manual.

Linker: –o (––output)

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Linker** entry and select **Output Format**.

3. Enable one or more output formats

Command line syntax

-o[*filename*][**:***format*[**:***addr_size*][,*space_name*]]...
––output=[*filename*][**:***format*[**:***addr_size*][,*space_name*]]...

You can specify the following formats:

IEEE	IEEE–695
ELF	ELF/DWARF
IHEX	Intel Hex
SREC	Motorola S–records

Description

By default, the linker generates an output file in ELF/DWARF format, named after the first input file with extension `.abs`.

With this option you can specify an alternative *filename*, and an alternative *output* format. The default output format is the format of the first input file.

You can use the **–o** option multiple times. This is useful to generate multiple output formats. With the first occurrence of the **–o** option you specify the basename (the filename without extension), which is used for subsequent **–o** options with no filename specified. If you do not specify a filename, or you do not specify the **–o** 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) or **4** bytes (S3 records, default).

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. The other address spaces are also emitted whereas their output files are named *`filename_spacename`*`.hex` (`.sre`). If you do not specify *space_name*, or you specify a non–existing space, the default address space is assumed.

Use option **–c** (**––chip–output**) 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 code:

```
tlk test.obj -omyfile.hex:IHEX,code
tlk test.obj --output=myfile.hex:IHEX,code
```

To create the output file `myfile.hex` of the default address space:

```
tlk test.obj -omyfile.hex:IHEX
tlk test.obj --output=myfile.hex:IHEX
```

If they exist, any other address spaces are emitted as well and are named `myfile_`*`spacename`*`.hex`.

Related information

Linker option **-c** (Generate an output file for each chip)

Linker: –r (––incremental)

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Linker** entry and select **Miscellaneous**.
3. Enable the option **Link incrementally**.

Command line syntax

–r
––incremental

Description

Normally the linker links and locates the specified object files. With this option you tell the linker only to link the specified files. The linker creates a linker output file `.out`. You then can link this file again with other object files until you have reached the final linker output file that is ready for locating.

In the last pass, you call the linker without this option with the final linker output file `.out`. The linker will now locate the file.

Example

In this example, the files `test1.obj`, `test2.obj` and `test3.obj` are incrementally linked:

1. `tlk -r test1.obj test2.obj -otest.out`

   `test1.obj` and `test2.obj` are linked

2. `tlk --incremental test3.obj test.out`

   `test3.obj` and `test.out` are linked, `task1.out` is created

3. `tlk task1.out`

   `task1.out` is located

Related information

Section 7.5, *Incremental Linking* in chapter *Using the Linker* of the user's manual.

Linker: –S (––strip–debug)

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Linker** entry and select **Miscellaneous**.
3. *Disable* the option **Include symbolic debug information**.

Command line syntax

–S
––strip–debug

Description

With this option you specify not to include symbolic debug information in the resulting output file.

Related information

–

Linker: –V (––version)

Menu entry

–

Command line syntax

–V
––version

Description

Display version information. The linker ignores all other options or input files.

Related information

–

Linker: –v/–vv (––verbose/––extra–verbose)

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Linker** entry and select **Miscellaneous**.

3. Add the option **––verbose** or **––extra–verbose** to the **Additional linker options** field.

Command line syntax

–v/–vv
––verbose/––extra–verbose

Description

With this option you put the linker in *verbose* mode. The linker prints the link phases while it processes the files. In the *extra verbose* mode, the linker also prints the filenames and it shows which objects are extracted from libraries. With this option you can monitor the current status of the linker.

Related information

–

Linker: –w (––no–warnings)

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Linker** entry and select **Diagnostics**.

3. Set **Error reporting** to one of the following values:

   - **Report all warnings**
   - **Suppress all warnings**
   - **Suppress specific warnings**.

   *If you select **Suppress specific warnings**:*

4. Enter the numbers, separated by commas, of the warnings you want to suppress.

Command line syntax

-w[*nr*]
––no–warnings[=*nr*]

Description

With this option you can suppresses all warning messages or specific warning messages.

- If you do not specify this option, all warnings are reported.
- If you specify this option but without numbers, all warnings are suppressed.
- If you specify this option with a number, only the specified warning is suppressed. You can specify the option **–w** multiple times.

Example

To suppress warnings 135 and 136, enter **135, 136** in the **Specific warnings to suppress** field, or enter the following on the command line:

```
tlk -w135,136 test.obj
tlk --no-warnings=135,136 test.obj
```

Related information

Linker option **––warnings–as–errors** (Treat warnings as errors)

Linker: ––warnings–as–errors

Menu entry

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

   The Project Options dialog box appears.

2. Expand the **Linker** entry and select **Diagnostics**.

3. Enable the option **Treat warnings as errors**.

Command line syntax

––warnings–as–errors

Description

With this option you tell the linker to treat warnings as errors.

When the linker detects an error or warning, it tries to continue the link process and reports other errors and warnings. With this option, the linker will exit with an exit status not equal zero (!= 0) for both errors and warnings and will not produce any output files.

Related information

Linker option **–w** (Suppress some or all warnings)

4.4 Control Program Options

The control program is a tool to facilitate use of the toolchain from the command line. Therefor you can only call the control program from the command line. The invocation syntax is:

```
tcc −T3000 [option]... [file]...
```

The option **−T3000** must always be specified to tell the control program to invoke the tools for the TSK3000.

Options

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 compiler, assembler or linker, it is recommended to use the control program options **−Wc**, **−Wa**, **−Wl**.

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 double minus (−−) characters. You can abbreviate long option names as long as the name is unique. You can mix short and long option names on the command line.

Options can have flags or sub−options. 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:

```
tcc −T3000 −Wc−Oac test.c
tcc −T3000 −−pass−c=−−optimize=+coalescer,+cse test.c
```

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

Control Program: –? (––help)

Command line syntax

–?
––help[=**options**]

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:

```
tcc -?
tcc --help
tcc
```

To see a detailed description of the available options, enter:

```
tcc --help=options
```

Control Program: --address-size

Command line syntax

--address-size=*addr_size*

Description

If you specify IHEX or SREC with the control option **--format**, you can additionally specify the record length and the address space to be emitted in the output files.

With this option you can specify the size of addresses in bytes (record length). For Intel Hex you can use the values: **1**, **2**, and **4** (default). For Motorola S-records you can specify: **2** (S1 records), **3** (S2 records, default) or **4** bytes (S3 records).

If you do not specify *addr_size*, the default address size is generated.

Example

To create the SREC file `test.sre` with S1 records, type:

```
tcc -T3000 --format=SREC --address-size=2 test.c
```

Related information

Control program option **--format** (Set linker output format)
Control program option **--space** (Set linker output space name)

Linker option **-o** (Specify an output object file)

Control Program: –cs/–co/–cl (––create)

Command line syntax

–cs
––create=assembly

–co
––create=object

–cl
––create=relocatable

Description

Normally the control program generates an absolute object file of the specified output format from the file you supplied as input.

With this option you tell the control program to stop after a certain number of phases.

–cs (**––create=assembly**)	Stop after C files are compiled to assembly (`.src`)
–co (**––create=object**)	Stop after the files are assembled to objects (`.obj`)
–cl (**––create=relocatable**)	Stop after the files are linked to a linker object file (`.out`)

Related information

Linker option **––link–only** (Link only, no locating)

Control Program: −−check

Command line syntax

−−check

Description

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

The compiler/assembler reports any warnings and/or errors.

Related information

Compiler option **−−check** (Check syntax)

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

Control Program: –D (––define)

Command line syntax

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

Description

With this option you can define a macro and specify it to the preprocessor. If you only specify a macro name (no macro definition), the macro expands as '1'.

You can specify as many macros as you like. On the command line, use the option **–D** multiple times. If the command line exceeds the length limit of the operating system, you can define the macros in an *option file* which you then must specify to the control program with the option **–f** *file*.

Defining macros with this option (instead of in the C source) is, for example, useful to compile or assemble conditional source as shown in the example below.

The control program passes the option **–D** (**––define**) to the compiler and the assembler.

Example

Consider the following C program with conditional code to compile a demo program and a real program:

```
void main( void )
{
#if DEMO == 1
    demo_func();   /* compile for the demo program */
#else
    real_func();   /* compile for the real program */
#endif
}
```

You can now use a macro definition to set the DEMO flag. With the control program this looks as follows:

```
tcc -T3000 -DDEMO test.c
tcc -T3000 -DDEMO=1 test.c

tcc -T3000 --define=DEMO test.c
tcc -T3000 --define=DEMO=1 test.c
```

Note that all four invocations have the same effect.

The next example shows how to define a macro with arguments. Note that the macro name and definition are placed between double quotes because otherwise the spaces would indicate a new option.

```
tcc -T3000 -D"MAX(A,B)=((A) > (B) ? (A) : (B))" test.c
```

Related information

Control Program option **–U** (Undefine preprocessor macro)
Control Program option **–f** (Read options from file)

Control Program: –d (––lsl–file)

Command line syntax

–d*file*
––lsl–file=*file*

Description

A linker script file contains vital information about the core for the locating phase of the linker. A linker script file is coded in LSL and contains the following types of information:

- the architecture and derivative definition describe the core's hardware architecture and its internal memory.
- the board specification describes the physical memory available in the system.
- the section layout definition describes how to locate sections in memory.

With this option you specify a linker script file via the control program to the linker. If you do not specify this option, the linker does not use a script file. You can specify the existing file `3000.lsl` or the name of a manually written linker script file. You can use this option multiple times. The linker processes the LSL files in the order in which they appear on the command line.

Related information

Section 7.7, *Controlling the Linker with a Script*, in chapter *Using the Linker* of the user's manual.

Control Program: −−diag

Command line syntax

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

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to `stdout` (normally your screen) and in the format you specify. You can specify the following formats: **html**, **rtf** or **text** (default). To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas.

With this option the control program does not process any files.

Example

To display an explanation of message number 103, enter:

```
tcc --diag=103
```

This results in message 103 with explanation. Note that the suboption **−T3000** is not necessary here, because the control program is not supposed to invoke any of the tools from the toolchain.

To write an explanation of all errors and warnings in HTML format to file `ccerrors.html`, enter:

```
tcc --diag=html:all > ccerrors.html
```

Related information

–

Control Program: –E (––preprocess)

Command line syntax

–E[*flags*]
––preprocess[=*flags*]

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

c/**C**(+/–**comments**)	Keep comments
p/**P**(+/–**noline**)	Strip #line source position info

Description

With this option you tell the control program to preprocess the C source.

The compiler sends the preprocessed file to stdout. To capture the information in a file, specify an output file with the option **–o**.

- With **–Ec** you tell the preprocessor to keep the comments from the C source file in the preprocessed output.
- With **–Ep** you tell the preprocessor to strip the #line source position information (lines starting with `#line`). These lines are normally processed by the assembler and not needed in the preprocessed output. When you leave these lines out, the output is more orderly to read.

Example

```
tcc –T3000 –EcP test.c –o test.pre
tcc –T3000 ––preprocess +comments,–noline test.c ––output=test.pre
```

The compiler preprocesses the file `test.c` and sends the output to the file `test.pre`. Comments are included but the line source position information is not stripped from the output file.

Related information

–

Control Program: ––error–file

Command line syntax

––error–file[=*file*]

Description

With this option the control program redirects error messages to a file.

If you do not specify a filename, the error file will be named after the input file with extension `.err` (for compiler) or `.ers` (for assembler).

Example

To write errors to `errors.err` instead of `stderr`, enter:

```
tcc -T3000 --error-file=errors.err test.c
```

Related information

Control Program option **––warnings–as–errors** (Treat warnings as errors)

Control Program: –F (––no–double)

Command line syntax

–F

––no–double

Description

With this option you tell the compiler to treat variables of the type `double` as `float`. Because the float type takes less space, execution speed increases and code size decreases, both at the cost of less precision.

Related information

–

Control Program: –f (––option–file)

Command line syntax

–f *file*
––option–file=*file*

Description

Instead of typing all options on the command line, you can create an option file which contains all options and 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 **–f** multiple times.

Format of an option file

- Multiple arguments on one line in the option file are allowed.
- To include whitespace in an argument, surround the argument with single or double quotes.
- If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

```
"This has a single quote ' embedded"

'This has a double quote " embedded'

'This has a double quote " and a single quote '"' embedded"
```

- When a text line reaches its length limit, use a 'to continue the line. Whitespace between quotes is preserved.

```
"This is a continuation \
line"

    -> "This is a continuation line"
```

- It is possible to nest command line files up to 25 levels.

Example

Suppose the file `myoptions` contains the following lines:

```
-T3000
-DDEMO=1
test.c
```

Specify the option file to the control program:

```
tcc -f myoptions
tcc --option-file=myoptions
```

This is equivalent to the following command line:

```
tcc -T3000 -DDEMO=1 test.c
```

Related information

–

Control Program: −−format

Command line syntax

−−format=*format*

You can specify the following formats:

IEEE	IEEE−695
ELF	ELF/DWARF
IHEX	Intel Hex
SREC	Motorola S−records

Description

With this option you specify the output format for the resulting (absolute) object file. The default output format is ELF/DWARF, which can directly be used by the debugger.

If you choose IHEX or SREC, you can additionally specify the address size of the chosen format (option **−−address−size**) and the address space to be emitted (option **−−space**)

Example

To generate an Motorola S−record output file:

```
tcc −T3000 −−format=SREC test1.c test2.c −−output=test.sre
```

Related information

Control program option **−−address−size** (Set address size for linker IHEX/SREC files)
Control program option **−−space** (Set linker output space name)

Linker option **−o** (Output format)
Linker option **−c** (Generate hex file)

Control Program: −−fp−trap

Command line syntax

−−fp−trap

Description

By default the control program uses the non−trapping floating−point library (`fp3000.lib`). With this option you tell the control program to use the trapping floating−point library (`fp3000t.lib`).

If you use the trapping floating−point library, exceptional floating−point cases are intercepted and can be handled separately by an application defined exception handler. Using this library decreases the execution speed of your application.

Related information

–

Control Program: –g (––debug–info)

Command line syntax

–g
––debug–info

Description

With this option you tell the control program to include debug information in the generated object file.

Related information

–

Control Program: –I (––include–directory)

Command line syntax

–I*path*,...
––include–directory=*path*,...

Description

With this option you can specify the path where your include files are located. A relative path will be relative to the current directory.

Example

Suppose that the C source file `test.c` contains the following lines:

```
#include <stdio.h>
#include ”myinc.h”
```

You can call the control program as follows:

```
tcc –T3000 –Imyinclude test.c
```

First the compiler looks for the file `stdio.h` in the directory `myinclude` relative to the current directory. If it was not found, the compiler searches in the environment variable and then in the default `include` directory.

The compiler now looks for the file `myinc.h`, in the directory where `test.c` is located. If the file is not there the compiler searches in the directory `myinclude`. If it was still not found, the compiler searches in the environment variable and then in the default `include` directory.

Related information

Compiler option **–I** (Add directory to include file search path)
Compiler option **–H** (Include file at the start of a compilation)

Section 4.5, *How the Compiler Searches Include Files*, in chapter *Using the Compiler* of the user’s manual.

Control Program: −−iso

Command line syntax

−−iso={**90**|**99**}

Description

With this option you specify to the control program against which ISO standard it should check your C source. C90 is also referred to as the ”ANSI C standard”. C99 refers to the newer ISO/IEC 9899:1999 (E) standard and is the default.

Independant of the chosen ISO standard, the control program always links libraries with C99 support.

Example

To compile the file `test.c` conform the ISO C90 standard:

```
tcc −T3000 −−iso=90 test.c
```

Related information

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

Control Program: –k (––keep–output–files)

Command line syntax

–k
––keep–output–files

Description

If an error occurs during the compilation, assembling or linking process, the resulting output file may be incomplete or incorrect. With this option you keep the generated output files when an error occurs.

By default the control program removes generated output files when an error occurs. This is useful when you use the make utility. If the erroneous files are not removed, the make utility may process corrupt files on a subsequent invocation.

Use this option when you still want to use the generated files. For example when you know that a particular error does not result in a corrupt file, or when you want to inspect the output file, or send it to Altium support.

Related information

–

Control Program: –L (––library–directory / ––ignore–default–library–path)

Command line syntax

–L*dir*
––library–directory=*dir*

–L
––ignore–default–library–path

Description

With this option you can specify the path(s) where your system libraries, specified with the **–l** option, are located. If you want to specify multiple paths, use the option **–L** for each separate path.

By default path this is `$(PRODDIR)\c3000\lib` directory.

If you specify only **–L** (without a pathname) or the long option **––ignore–default–library–path**, the linker will not search the default path and also not in the paths specified in the environment variable `LIBTSK3000`. So, the linker ignores steps 2 and 3 as listed below.

The priority order in which the linker searches for system libraries specified with the **–l** option is:

1. The path that is specified with the **–L** option.
2. The path that is specified in the environment variable `LIBTSK3000`.
3. The default directory `$(PRODDIR)\c3000\lib` (or a processor specific sub–directory).

Example

Suppose you call the control program as follows:

```
tcc –T3000 test.c –Lc:\mylibs –lc3000
tcc –T3000 test.c ––library–directory=c:\mylibs ––library=c3000
```

First the linker looks in the directory `c:\mylibs` for library `c3000.lib` (this option).

If it does not find the requested libraries, it looks in the directory that is set with the environment variable `LIBTSK3000`.

Then the linker looks in the default directory `$(PRODDIR)\c3000\lib` for libraries.

Related information

Linker option **–l** (Link system library)

Control Program: –l (––library)

Command line syntax

–l*name*
––library=*name*

Description

With this option you tell the linker via the control program to use system library `name.lib`, where *name* is a string. The linker first searches for system libraries in any directories specified with **–L***path*, then in the directories specified with the environment variable `LIBTSK3000`, unless you used the option **–L** without a directory.

Example

To search in the system library `c3000.lib` (C library):

```
tcc –T3000 test.obj mylib.lib –lc3000
tcc –T3000 test.obj mylib.lib ––library=c3000
```

The linker links the file `test.obj` and first looks in `mylib.lib` (in the current directory only), then in the system library `c3000.lib` to resolve unresolved symbols.

Related information

Linker option **–L** (Additional search path for system libraries)

Section 7.4, *Linking with Libraries*, in chapter *Using the Linker* of the user's manual.

Control Program: --list-files

Command line syntax

--list-files[=*name*]

Description

With this option you tell the assembler via the control programma 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 *name* 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 *name*, or you specify more than one input files, the control program names the generated list file(s) after the specified input file(s) with extension `.lst`.

Example

This example generates the list files `1.lst` and `2.lst` for `1.c` and `2.c`. If in this example also a *name* had been specified, it would be ignored because two input files are specified.

```
tcc -T3000 1.c 2.c --list-files
```

Related information

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

Assembler option **-L** (List file formatting options)

Control Program: –n (––dry–run)

Command line syntax

–n
––dry–run

Description

With this option you put the control program *verbose* mode. The control program prints the invocations of the tools it would use to process the files without actually performing the steps.

Related information

Control Program option **–v** (**––verbose**) (Verbose output)

Control Program: −−no−default−libraries

Command line syntax

−−no−default−libraries

Description

By default the control program specifies the standard C libraries (C99) and run−time library to the linker. With this option you tell the control program *not* to specify the standard C libraries and run−time library to the linker.

In this case you must specify the libraries you want to link to the linker with the option **−l***library_name*. The control program recognizes the option **−l** as an option for the linker and passes it as such.

Example

```
tcc −T3000 −−no−default−libraries test.c
```

The control program does not specify any libraries to the linker. In normal cases this would result in unresoved externals.

To specify your own libraries (`libmy.a`) and avoid unresolved externals:

```
tcc −T3000 −−no−default−libraries −lmy test.c
```

Related information

Linker option **−l (−−library)** (Add library)

Control Program: ––no–map–file

Command line syntax

––no–map–file

Description

By default the control program tells the linker to generate a linker map file.

A linker map file is a text file that shows how the linker has mapped the sections and symbols from the various object files (`.obj`) to the linked object file. A locate part shows the absolute position of each section. External symbols are listed per space with their absolute address, both sorted on symbol and sorted on address.

With this option you prevent the generation of a map file.

Related information

–

Control Program: –o (––output–file)

Command line syntax

–o *file*
––output–file=*file*

Description

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

```
tcc -T3000 test.c prog.c
```

The control program generates an ELF/DWARF object file (default) with the name `test.abs`.

To generate the file `result.abs`:

```
tcc -T3000 -o result.abs test.c prog.c
tcc -T3000 --output-file=result.abs test.c prog.c
```

Related information

–

Control Program: –p (––profile)

Command line syntax

–p[*flags*]
––profile[=*flags*]

Use the following option for a predefined set of flags:

–pg	(**––profile=g**)	profiling with call graph and function timers Alias for: **–pBcFt**

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

b/**B**	(+/–**block**)	**block counters**
c/**C**	(+/–**callgraph**)	**call graph**
f/**F**	(+/–**function**)	**function counters**
t/**T**	(+/–**time**)	**function timers**

Description

Profiling is the process of collecting statistical data about a running application. With these data you can analyze which functions are called, how often they are called and what their execution time is.

Several methods of profiling exists. One method is *code instrumentation* which adds code to your application that takes care of the profiling process when the application is executed.

For an extensive description of profiling refer to Chapter 5, *Profiling* in the user's manual.

With this option, the compiler adds the extra code to your application that takes care of the profiling process. You can obtain the following profiling data (see flags above):

Block counters (not in combination with Call graph or Time)

This will instrument the code to perform basic block counting. As the program runs, it will count how many time it executed each branch of each `if` statement, each iteration of a `for` loop, and so on. Note that though you can combine Block counters with Function counters, this has no effect because Function counters is only a subset of Block counters.

Call graph (not in combination with Block counters)

This will instrument the code to reconstruct the run–time call graph. As the program runs it associates the caller with the gathered profiling data.

Function counters

This will instrument the code to perform function call counting. This is a subset of the basic Block counters.

Time (not in combination with Block counters)

This will instrument the code to measure the time spent in a function. This includes the time spend in all called functions (callees).

Note that the more detailled information you request, the larger the overhead in terms of execution time, code size and heap space needed. The option **Generate Debug information** (**–g** or **––debug**) does not affect profiling, execution time or code size.

The control program automatically links with the corresponding profiling librabries.

Example

To generate block count information for the module `test.c` during execution, compile as follows:

```
tcc –T3000 –pb test.c
tcc –T3000 ––profile=+block test.c
```

In this case you must link the library `pb3000.lib`.

Related information

Chapter 5, *Profiling* in the user's manual.

```
tcc –T3000 ––format=IHEX ––space=code test.c
```

Control Program: ––static

Command line syntax

––static

Description

This option is directly passed to the compiler.

With this option, the compiler treats external definitions at file scope (except for `main`) as if they were declared `static`. As a result, unused functions will be eliminated, and the alias checking algorithm assumes that objects with static storage cannot be referenced from functions outside the current module.

This option only makes sense when you specify all modules of an application on the command line.

Example

```
tcc -T3000 --static module1.c module2.c module3.c
```

Related information

–

Control Program: –T (––target)

Command line syntax

–T*target*
––target=*target*

You must specify the following target:

3000	for the TSK3000

Description

You must always specify this option. With this option you specify the target to the control program. Based on this setting, the control program invokes the tools for the chosen target.

Related information

–

Control Program: –t (––keep–temporary–files)

Menu Entry

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

   The Project Options dialog box appears.

5. Select **Build Options**.

6. Enable the option **Keep temporary files that are generated during a compile**.

Command line syntax

–t
––keep–temporary–files

Description

By default, the control program removes intermediate files like the `.src` file (result of the compiler phase) and the `.obj` file (result of the assembler phase).

With this option you tell the control program to keep temporary files it generates during the creation of the absolute object file.

Related information

–

Control Program: –U (––undefine)

Command line syntax

–U*macro_name*
––undefine=*macro_name*

Description

With this option you can undefine an earlier defined macro as with `#undef`.

This option is for example useful to undefine predefined macros. However, you cannot undefine predefined ISO C standard macros.

The control program passes the option **–U** (**––undefine**) to the compiler.

Example

To undefine the predefined macro __TASKING__:

```
tcc -T3000 -U__TASKING__ test.c
tcc -T3000 --undefine=__TASKING__ test.c
```

Related information

Control Pogram option **–D** (Define preprocessor macro)

Control Program: –V (––version)

Command line syntax

–V
––version

Description

Display version information. The control program ignores all other options or input files.

Related information

–

Control Program: –v (––verbose)

Command line syntax

–v
––verbose

Description

With this option you put the control program in verbose mode. With the option **–v** the control program performs it tasks while it prints the steps it performs to `stdout`.

Related information

Control Program option **–n** (**––dry–run**) (Verbose output and suppress execution)

Control Program: –W (––pass)

Command line syntax

–Wc*option*	**––pass–c=***option*	Pass option directly to the C compiler
–Wa*option*	**––pass–assembler=***option*	Pass option directly to the assembler
–Wl*option*	**––pass–linker=***option*	Pass option directly to the linker

Description

With this option you tell the control program to call a tool with the specified option. The control program does not use or interpret the option itself, but specifies it directly to the tool which it calls.

Related information

–

Control Program: –w (––no–warnings)

Command line syntax

–w[*nr*]
––no–warnings[=*nr*]

Description

With this option you can suppress all warning messages or specific C compiler warning messages:

- If you do not specify this option, all warnings are reported.
- If you specify this option but without numbers, all warnings are suppressed.
- If you specify this option with a number, only the specified warning is suppressed.
  You can specify the option **–w** multiple times.

Related information

–

Control Program: ––warnings–as–errors

Command line syntax

––warnings–as–errors

Description

With this option you tell the control program to treat warnings as errors.

If one of the tools encounters an error, it stops processing the file(s). With this option the tools treat warnings as errors and therefor will continue processing the files, even in case of errors.

Related information

Control Program option **–w** (Suppress all warnings)

4.5 Make Utility Options

When you build a project in Altium Designer, Altium Designer generates a makefile and uses the make utility **tmk** 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:

```
tmk [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 Altium Designer.

Defining Macros

Command line syntax

macro=*definition*

Description

With this argument you can define a macro and specify it to the make utility.

A macro definition remains in existence during the execution of the makefile, even when the makefile recursively calls the make utility again. In the recursive call, the macro acts as an environment variable. This means that it is overruled by definitions in the recursive call. Use the option **–e** to prevent this.

You can specify as many macros as you like. If the command line exceeds the limit of the operating system, you can define the macros in an *option file* which you then must specify to the 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.abs : demo.obj main.obj
             tlk demo.obj main.obj -d3000.lsl -lc3000 -lfp3000
else
  real.abs : real.obj main.obj
             tlk real.obj main.obj -d3000.lsl -lc3000 -lfp3000
endif
```

You can now use a macro definition to set the `DEMO` flag:

```
tmk real.abs DEMO=1
```

In both cases the absolute object file `real.abs` is created but depending on the DEMO flag it is linked with `demo.obj` or with `real.obj`.

Related information

Make utility option **–e** (Environment variables override macro definitions)
Make utility option **–m** (Name of invocation file)

Make Utility: –?

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:

```
tmk -?
```

Related information

–

Make Utility: −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

```
tmk −a
```

Rebuilds all your files, regardless of whether they are out of date or not.

Related information

–

Make Utility: –c

Command line syntax

–c

Description

Altium Designer uses this option for the graphical version of the make utility 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

```
tmk –c
```

The make utility runs its commands as a child processes.

Related information

–

Make Utility: –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 **tmk**.

With the option **–DD** not only the lines of the makefile are printed but also the lines of the `tmk.mk` file (implicit rules).

Example

```
tmk –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: −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

```
tmk −d
```

Shows which files are out of date and rebuilds them.

Related information

−

Make Utility: −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

```
tmk -e
```

The make utility uses the settings of the environment variables regardless of macro definitions.

Related information

–

Make Utility: −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

```
tmk -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: –f

Command line syntax

–f *my_makefile*

Description

Default the make utility uses the file `makefile` to build your files.

With this option you tell the make utility to use the specified file instead of the file `makefile`. Multiple **–f** options act as if all the makefiles were concatenated in a left–to–right order.

Example

```
tmk -f mymake
```

The make utility uses the file `mymake` to build your files.

Related information

–

Make Utility: –G

Command line syntax

–G *path*

Description

Normally you must call the make utility **tmk** 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:

```
tmk –G ..\myfiles
```

Related information

–

Make Utility: –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

```
tmk –i
```

The make utility exits without an error code, even when an error occurs.

Related information

–

Make Utility: –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

```
tmk -K
```

The make utility preserves all temporary files.

Related information

–

Make Utility: −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

```
tmk -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: –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.

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:

```
-k
-err errors.txt
test.abs
```

Specify the option file to the make utility:

```
tmk -m myoptions
```

This is equivalent to the following command line:

```
tmk -k -err errors.txt test.abs
```

Related information

–

Make Utility: −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

```
tmk -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: −p

Command line syntax

−q

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 dependency files are never removed.

Example

```
tmk −p
```

The make utility never removes target dependency files.

Related information

–

Make Utility: −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

```
tmk −q
```

The make utility only returns an exit code that indicates whether all target files are up to date or not. It does not rebuild any files.

Related information

−

Make Utility: –r

Command line syntax

–r

Description

When you call the make utility, it first reads the implicit rules from the file `tmk.mk`, then it reads the makefile with the rules to build your files. (The file `tmk.mk` is located in the `\etc` directory of the toolchain.)

With this option you tell the make utility *not* to read `tmk.mk` and to rely fully on the make rules in the makefile.

Example

```
tmk -r
```

The make utility does not read the implicit make rules in `tmk.mk`.

Related information

–

Make Utility: −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.

Example

```
tmk -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 **tmk** in the makefile.

Related information

Make utility option **−k** (On error, abandon the work for the current target only)

Make Utility: –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

```
tmk -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: –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

```
tmk -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: –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

```
tmk -time
```

The make utility displays the current date and time and updates out–of–date files.

Related information

–

Make Utility: –V

Command line syntax

–V

Description

Display version information. The make utility ignores all other options or input files.

Example

```
tmk -V
```

The make utility displays the version information but does not perform any tasks.

Related information

–

Make Utility: –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

```
tmk -W test.abs
```

The make utility rebuilds out of date targets in the makefile except the file `test.abs` which is considered now as up to date.

Related information

–

Make Utility: −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. Altium Designer uses this option for the graphical version of make.

Example

```
tmk −x
```

If errors occur, the make utility gives extended information.

Related information

−

4.6 Librarian Options

The librarian **tlb** is a tool to build library files and it offers the possibility to replace, extract and remove modules from an existing library.

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

tlb *key_option* [*sub_option*...] *library* [*object_file*]

This section describes all options for the make utility. Suboptions can only be used in combination with certain key options. Keyoptions and their suboptions are therefor described together. The miscellaneous options can always be used and are also described separately.

The librarian is a command line tool so there are no equivalent options in Altium Designer.

Description	Option	Suboption
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**	
Suboptions		
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		
Display options	**–?**	
Display version header	**–V**	
Read options from *file*	**–f** *file*	
Suppress warnings above level *n*	**–w***n*	

Table 4–1: Overview of librarian options and suboptions

Librarian: –?

Command line syntax

–?

Description

Displays an overview of all command line options.

Example

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

```
tlb -?
tlb
```

Related information

–

Librarian: −d

Command line syntax

−d [−**v**]

Description

Delete the specified object modules from a library. With the suboption **−v** the librarian shows which files are removed.

−v Verbose: the librarian shows which files are removed.

Example

```
tlb −d mylib.lib obj1.obj obj2.obj
```

The librarian deletes `obj1.obj` and `obj2.obj` from the library `mylib.lib`.

```
tlb −d −v mylib.lib obj1.obj obj2.obj
```

The librarian deletes `obj1.obj` and `obj2.obj` from the library `mylib.lib` and displays which files are removed.

Related information

–

Librarian: –f

Command line syntax

–f *file*

Description

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

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

Option files can also be generated on the fly, for example by the make utility. You can specify the option **–f** multiple times.

Format of an option file

- Multiple arguments on one line in the option file are allowed.
- To include whitespace in an argument, surround the argument with single or double quotes.
- If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

```
"This has a single quote ' embedded"

'This has a double quote " embedded'

'This has a double quote " and a single quote '"' embedded"
```

- When a text line reaches its length limit, use a ’to continue the line. Whitespace between quotes is preserved.

```
"This is a continuation \
line"
     -> "This is a continuation line"
```

- It is possible to nest command line files up to 25 levels.

Example

Suppose the file `myoptions` contains the following lines:

```
-x mylib.lib obj1.obj
-w5
```

Specify the option file to the librarian:

```
tlb -f myoptions
```

This is equivalent to the following command line:

```
tlb -x mylib.lib obj1.obj -w5
```

Librarian: –m

Command line syntax

–m [**–a** *posname*] [**–b** *posname*]

Description

Move the specified object modules to another position in the library.

The ordering of members in a library can make a difference in how programs are linked if a symbol is defined in more than one member.

Default, the specified members are moved to the end of the archive. Use the suboptions **–a** or **–b** to move them to a specified place instead.

–a *posname*	Move the specified object module(s) after the existing module *posname*.
–b *posname*	Move the specified object module(s) before the existing module *posname*.

Example

Suppose the library `mylib.lib` contains the following objects (see option **–t**):

```
obj1.obj
obj2.obj
obj3.obj
```

To move `obj1.obj` to the end of `mylib.lib`:

```
tlb -m mylib.lib obj1.obj
```

To move `obj3.obj` just before `obj2.obj`:

```
tlb -m -b obj3.obj mylib.lib obj2.obj
```

The library `mylib.lib` after these two invocations now looks like:

```
obj3.obj
obj2.obj
obj1.obj
```

Related information

Librarian option **–t** (Print library contents)

Librarian: −p

Command line syntax

−p

Description

Print the specified object module(s) in the library to standard output.

This option is only useful when you redirect or pipe the output to other files or tools that serve your own purposes. Normally you do not need this option.

Example

```
tlb −p mylib.lib obj1.obj > file.obj
```

The librarian prints the file `obj1.obj` to standard output where it is redirected to the file `file.obj`. The effect of this example is very similar to extracting a file from the library but in this case the ’extracted’ file gets another nam.

Related information

–

Librarian: −r

Command line syntax

−r [**−a** *posname*] [**−b** *posname*] [**−c**] [**−u**] [**−v**]

Description

You can use the option **−r** for several purposes:

- Adding new objects to the library
- Replacing objects in the library with the same object of a newer date
- Creating a new library

The option **−r** normally *adds* a new module to the library. However, if the library already contains a module with the specified name, the existing module is *replaced*. If you specify a library that does not exist, the librarian *creates* a new library with the specified name.

If you add a module to the library without specifying the suboption **−a** or **−b**, the specified module is added at the end of the archive. Use the suboptions **−a** or **−b** to insert them to a specified place instead.

−a *posname*	Add the specified object module(s) after the existing module *posname*.
−b *posname*	Add the specified object module(s) before the existing module *posname*.
−c	Create a new library without checking whether it already exists. If the library already exists, it is overwritten.
−u	Insert the specified object module only if it is newer than the module in the library.
−v	Verbose: the librarian shows which files are removed.

The suboptions **−a** or **−b** have no effect when an object is added to the library.

Examples

Suppose the library `mylib.lib` contains the following objects (see option **−t**):

```
obj1.obj
```

To add `obj2.obj` to the end of `mylib.lib`:

```
tlb −r mylib.lib obj2.obj
```

To insert `obj3.obj` just before `obj2.obj`:

```
tlb −r −b obj2.obj mylib.lib obj3.obj
```

The library `mylib.lib` after these two invocations now looks like:

```
obj1.obj
obj3.obj
obj2.obj
```

Creating a new library

To *create a new library file*, add an object file and specify a library that does not yet exist:

```
   tlb -r obj1.obj newlib.lib
```

The librarian creates the library `newlib.lib` and adds the object `obj1.obj` to it.

To *create a new library file and overwrite an existing library*, add an object file and specify an existing library with the supoption **−c**:

```
   tlb -r -c obj1.obj mylib.lib
```

The librarian overwrites the library `mylib.lib` and adds the object `obj1.obj` to it. The new library `mylib.lib` only contains `obj1.obj`.

Related information

Librarian option **−t** (Print library contents)

Librarian: –t

Command line syntax

–t [**–s0**|**–s1**]

Description

Print a table of contents of the library to standard out. With the suboption **–s** you the librarian displays all symbols per object file.

–s0	Displays per object the library in which it resides, the name of the object itself and all symbols in the object.
–s1	Displays only the symbols of all object files in the library.

Example

```
tlb -t mylib.lib
```

The librarian prints a list of all object modules in the libary `mylib.lib`.

```
tlb -t -s0 mylib.lib
```

The librarian prints per object all symbols in the library. This looks like:

```
prolog.obj
   symbols:
mylib.lib:prolog.obj:___Qabi_callee_save
mylib.lib:prolog.obj:___Qabi_callee_restore
div16.obj
   symbols:
mylib.lib:div16.obj:___udiv16
mylib.lib:div16.obj:___div16
mylib.lib:div16.obj:___urem16
mylib.lib:div16.obj:___rem16
```

Related information

–

Librarian: –V

Command line syntax

–V

Description

Display version information. The librarian ignores all other options or input files.

Example

```
tlb -V
```

The librarian displays version information but does not perform any tasks.

Related information

–

Librarian: –w

Command line syntax

–w*level*

Description

With this suboption you tell the librarian to suppress all warnings above the specified level. The level is a number between 0 – 9.

The level of a message is printed between parentheses after the warning number. If you do not use the **–w** option, the default warning level is 8.

Example

To suppresses warnings above level 5:

```
tlb -x -w5 mylib.lib obj1.obj
```

Related information

–

Librarian: −x

Command line syntax

−x [**−o**] [**−v**]

Description

Extract an existing module from the library.

−o Give the extracted object module the same date as the last−modified date that was recorded in the library.

Without this suboption it receives the last−modified date of the moment it is extracted.

−v Verbose: the librarian shows which files are extracted.

Examples

To extract the file `obj1.obj` from the library `mylib.lib`:

```
tlb −x mylib.lib obj1.obj
```

If you do not specify an object module, all object modules are extracted:

```
tlb −x mylib.lib
```

Related information

−

5 List File Formats

Summary

This chapter describes the format of the assembler list file and the linker map file.

5.1 Assembler List File Format

The assembler list file is an additional output file of the assembler that contains information about the generated code.

The list file consists of a page header and a source listing.

Page header

The page header is repeated on every page:

```
TASKING target Assembler vx.yrz Build nnn SN 00000000
Title                                                      Page 1

ADDR CODE      CYCLES  LINE SOURCE LINE
```

The first line contains version information.

The second line can contain a title which you can specify with the assembler directive `.TITLE` and always contains a page number. With the assembler directives `.LIST`/`.NOLIST` and `.PAGE`, and with the assembler option **–L***flag* (**––list–format**) you can format the list file.

See Section 3.2, *Assembler Directives* in Chapter *Assembly Language* and Section 4.2, *Assembler Options* in Chapter *Tools Options*.

The fourth line contains the headings of the columns for the source listing.

Source listing

The following is a sample part of a listing. An explanation of the different columns follows below.

```
   ADDR CODE       CYCLES   LINE SOURCE LINE
                               1         ; Module start
                               .
                               .
   0000 7Frr       2    2     20         mov      R7,#@lsb(__1_ini)
   0002 7Err       2    4     21         mov      R6,#@msb(__1_ini)
   0004 02rrrr     4    8     22         gjmp     _printf
                               .
                               .
   0000                       38         .ds      2
     |  RESERVED
   0001
```

The meaning of the different columns is:

ADDR	This column contains the memory address. The address is a hexadecimal number that represents the offset from the beginning of a relocatable section or the absolute address for an absolute section. The address only appears on lines that generate object code.
CODE	This is the object code generated by the assembler for this source line, displayed in hexadecimal format. The displayed code need not be the same as the generated code that is entered in the object module. The code can also be relocatable code. In this case the letter 'r' is printed for the relocatable code part in the listing. For lines that allocate space, the code field contains the text "RESERVED". For lines that initialize a buffer, the code field lists one value followed by the word "REPEATS".
CYCLES	The first number in this column is the number of instruction cycles needed to execute the instruction(s) as generated in the CODE field. The second number is the accumulated cycle count of this section.
LINE	This column contains the line number. This is a decimal number indicating each input line, starting from 1 and incrementing with each source line.
SOURCE LINE	This column contains the source text. This is a copy of the source line from the assembly source file.

For the .SET and .EQU directives the ADDR and CODE columns do not apply. The symbol value is listed instead.

Related information

See section 6.6, *Generating a List File*, in Chapter *Using the Assembler* of the user's manual for more information on how to generate a list file and specify the amount of list file information.

5.2 Linker Map File Format

The linker map file is an additional output file of the linker that shows how the linker has mapped the sections and symbols from the various object files (`.obj`) to output sections. The locate part shows the absolute position of each section. External symbols are listed per space with their absolute address, both sorted on symbol and sorted on address.

With the linker option **–m** (map file formatting) you can specify which parts of the map file you want to see.

Example (part of) linker map file

```
*********************************  Processed Files Part  ******************************
+-----------------------------------------------------------+
| File       | From archive | Symbol causing the extraction |
|===========================================================|
| cstart.obj | c3000.lib    | _Exit                         |
| hello.obj  |              |                               |
| printf.obj | c3000.lib    | printf                        |
+-----------------------------------------------------------+


***************************************  Link Part  ***********************************
+-------------------------------------------------------------------------------------+
| [in] File  | [in] Section | [in] Size  | [out] Offset | [out] Section | [out] Size |
|=====================================================================================|
| hello.obj  | .text        | 0x00000010 | 0x00000000   | .text         | 0x00000010 |
| printf.obj | .text        | 0x00000058 | 0x00000000   |               | 0x00000058 |
|-------------------------------------------------------------------------------------|
| cstart.obj | .text.cstart | 0x000000a0 | 0x00000000   | .text.cstart  | 0x000000a0 |
+-------------------------------------------------------------------------------------+


********************************  Module Local Symbols Part  **************************

* Symbol translation (sorted on name)
=====================================

+ Scope "./hello.obj"
+---------------------------------------------+
| Name    | Address    | Space                |
|=============================================|
| hello.c | 0x00000000 | -                    |
| .rodata | 0x000000a0 | TSK3000:TSK3000:main |
| .rodata | 0x000000a8 |                      |
| .sdata  | 0x00010000 |                      |
| .text   | 0x000000d8 |                      |
+---------------------------------------------+
***********************************  Cross Reference Part  ******************************

+-------------------------------------------------------------------------------+
| Definition file | Definition section  | Symbol     | Referenced in           |
|===============================================================================|
| cstart.obj      | .text.cstart        | _START     | hello.obj               |
| hello.obj       | .text               | main       | cstart.obj              |
+-------------------------------------------------------------------------------+

* Undefined symbols:
====================
+-----------------------+
| Symbol | Referenced in |
|=======================|
| __init | hello.obj     |
+-----------------------+


***************************************  Locate Part  ***********************************

* Task entry address
=====================
+------------------------------+
| symbol           | _START     |
| absolute address | 0x00000000 |
+------------------------------+

* Section translation
======================

+ Space TSK3000:TSK3000:main

+--------------------------------------------------------------------+
| Chip | Group | Section      | Size (MAU) | Space addr | Chip addr  |
|====================================================================|
| irom |       | .text.cstart | 0x000000a0 | 0x00000000 | 0x00000000 |
|      |       | .rodata      | 0x0000000c | 0x000000b4 | 0x000000b4 |
|      |       | [.data]      | 0x000000c8 | 0x00001978 | 0x00001978 |
|      |       | [.sdata]     | 0x00000004 | 0x00001a40 | 0x00001a40 |
|      |       | table        | 0x00000034 | 0x00001a44 | 0x00001a44 |
| iram | sda   | .sdata       | 0x00000004 | 0x00010000 | 0x00000000 |
|+-------------------------------------------------------------------+

* Symbol translation (sorted on name)
======================================
+----------------------------------------------------+
| Name           | Address    | Space                |
|====================================================|
| EXCEPTION_BASE | 0x00000100 | TSK3000:TSK3000:main |
| _Exit          | 0x00000098 |                      |
| _START         | 0x00000000 |                      |
| main           | 0x000000d8 |                      |
+----------------------------------------------------+
* Symbol translation (sorted on address)
========================================
+----------------------------------------------------+
| Address    | Name           | Space              |
|====================================================|
| 0x00000000 | _START         | TSK3000:TSK3000:main |
| 0x00000098 | _Exit          |                    |
| 0x000000d8 | main           |                    |
| 0x00000100 | EXCEPTION_BASE |                    |
+----------------------------------------------------+

***************************************  Memory Part  ***********************************

* Address range usage at space level
====================================
+--------------------------------------------------------------------------------------+
| Name                 | Total      | Used       %  | Free       %  | > free gap  %  |
|======================================================================================|
| TSK3000:TSK3000:main | 0x00080000 | 0x00080000 100 | 0x00000000   0 | 0x00000000   0 |
+--------------------------------------------------------------------------------------+

* Address range usage at memory level
=====================================
+----------------------------------------------------------------------+
| Name | Total      | Used       %  | Free       %  | > free gap  %  |
|======================================================================|
| iram | 0x00080000 | 0x00080000 100 | 0x00000000   0 | 0x00000000   0 |
| irom | 0x00010000 | 0x00001a95  11 | 0x0000e56b  89 | 0x0000e568  89 |
| xrom | 0x00080000 | 0x00000000   0 | 0x00080000 100 | 0x00080000 100 |
+----------------------------------------------------------------------+

*********************************  Linker Script File Part  *****************************


************************************  Locate Rule Part  *********************************
+-----------------------------------------------------------------------------------------+
| Address space        | Type          | Properties | Sections                            |
|=========================================================================================|
| TSK3000:TSK3000:main | absolute      | 0x00000000 | .text.cstart                        |
| TSK3000:TSK3000:main | contiguous    |            | .sdata | .sbss                      |
| TSK3000:TSK3000:main | unrestricted  |            | stack                               |
| TSK3000:TSK3000:main | unrestricted  |            | .bss   .data   xvwbuffer            |
| TSK3000:TSK3000:main | unrestricted  |            | .rodata   [.data]   [.sdata]   table |
| TSK3000:TSK3000:main | ballooned     |            | stack                               |
+-----------------------------------------------------------------------------------------+
```

The meaning of the different parts is:

Processed Files Part

This part of the map file shows all processed files. This also includes object files that are extracted from a library, with the symbol that led to the extraction.

Link Part

This part of the map file shows per object file how the link phase has mapped the sections from the various object files (`.obj`) to output sections.

`[in] File`	The name of an input object file.

`[in] Section`	A section name from the input object file.
`[in] Size`	The size of the input section.
`[out] Offset`	The offset relative to the start of the output section.
`[out] Section`	The resulting output section name.
`[out] Size`	The size of the output section.

Module Local Symbols Part

This part of the map file shows a table for each local scope within an object file. Each table has three columns, 1 the 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 **-mq** (module local symbols).

Cross Reference Part

This part of the map file lists all symbols defined in the object modules and for each symbol the object modules that contain a reference to the symbol are shown. Also, symbols that remain undefined are shown.

Locate Part: Section translation

This part of the map file shows the absolute position of each section in the absolute object file. It is organized per address space, memory chip and group and sorted on space address.

`+ Space`	The names of the address spaces as defined in the linker script file (`*.lsl`). The names are constructed of the `derivative` name followed by a colon ':', the `core` name, another colon ':' and the `space` name.
`Chip`	The names of the memory chips as defined in the linker script file (`*.lsl`) in the `memory` definitions.
`Group`	Sections can be ordered in groups. These are the names of the groups as defined in the linker script file (`*.lsl`) with the keyword `group` in the `section_layout` definition. The name that is displayed is the name of the deepest nested group.
`Section`	The name of the section. Names within square brackets **[ ]** will be copied during initialization from ROM to the corresponding section name in RAM.
`Size (MAU)`	The size of the section in minimum addressable units.
`Space addr`	The absolute address of the section in the address space.
`Chip addr`	The absolute offset of the section from the start of a memory chip.

Locate Part: Symbol translation

This part of the map file lists all external symbols per address space name, both sorted on address and sorted on symbol name.

`Name`	The name of the symbol.
`Address`	The absolute address of the symbol in the address space.
`Space`	The names of the address spaces as defined in the linker script file (`*.lsl`). The names are constructed of the `derivative` name followed by a colon ':', the `core` name, another colon ':' and the `space` name.

Memory Part

This part of the map file shows the memory usage in totals and percentages for spaces and chips. The largest free block of memory per space and per chip is also shown.

Linker Script File Part

This part of the map file shows the processor and memory information of the linker script file.

By default this part is not shown in the map file. You have to turn this part on manually with linker option **-ms** (processor and memory info). You can print this information to a separate file with linker option **--lsl-dump**.

Locate Rule Part

This part of the map file shows the rules the linker uses to locate sections.

`Address space`	The names of the address spaces as defined in the linker script file (`*.lsl`). The names are constructed of the `derivative` name followed by a colon ':', the `core` name, another colon ':' and the `space` name.
`Type`	The rule type:

`ordered/contiguous/clustered/unrestricted`	Specifies how sections are grouped. By default, a group is 'unrestricted' which means that the linker has total freedom to place the sections of the group in the address space.
`absolute`	The section must be located at the address shown in the Properties column.
`address range`	The section must be located in the union of the address ranges shown in the Properties column; end addresses are not included in the range.
`address range size`	The sections must be located in some address range with size not larger than shown in the Properties column; the second number in that field is the alignment requirement for the address range.
`ballooned`	After locating all sections, the largest remaining gap in the space is used completely for the stack and/or heap.

`Properties`	The contents depends on the Type column.
`Sections`	The sections to which the rule applies; restrictions between sections are shown in this column:

<	ordered
|	contiguous
+	clustered

For contiguous sections, the linker uses the section order as shown here. Clustered sections can be located in any relative order.

Related information

Section 7.9, *Generating a Map File*, in Chapter *Using the Linker* of the user's manual.
Linker option **–M** (Generate map file)

6 Object File Formats

Summary

This chapter describes the formats of several object files.

6.1 ELF/DWARF Object Format

The TASKING TSK3000 toolchain by default produces objects in the ELF/DWARF 2 format.

The ELF/DWARF 2 Object Format for the TSK3000 toolchain follows the convention as described in the *System V Application Binary Interface, MIPS RISC Processor Supplement 3rd Edition* [1990–1996, The Santa Cruz Operation, Inc.]

For a complete description of the ELF and DWARF formats, please refer to the *Tool Interface Standard (TIS)*.

6.2 Motorola S–Record Format

With the linker option **–o***filename***:SREC** option the linker produces output in Motorola S–record format with three types of S–records: S0, S3 and S7. With the options **–o***filename***:SREC:2** or **–o***filename***:SREC:3** option you can force other types of S–records. They have the following layout:

S0 – record

’S’ ’0’ <*length_byte*> <2 bytes 0> <*comment*> <*checksum_byte*>

A linker generated S–record file starts with a S0 record with the following contents:

length_byte	: $06
comment	: tlk
checksum	: $AE

```
        t l k
S0060000746C6BAE
```

The S0 record is a comment record and does not contain relevant information for program execution.

The length_byte represents the number of bytes in the record, not including the record type and length byte.

The checksum is calculated by first adding the binary representation of the bytes following the record type (starting with the length_byte) to just before the checksum. Then the one’s complement is calculated of this sum. The least significant byte of the result is the checksum. The sum of all bytes following the record type is 0FFH.

S1 – record

With the linker option **–o***filename***:SREC:2**, the actual program code and data is supplied with S1 records, with the following layout:

’S’ ’1’ <*length_byte*> <*address*> <*code bytes*> <*checksum_byte*>

This record is used for 2–byte addresses.

Example:

```
S1130250F03EF04DF0ACE8A408A2A013EDFCDB00E6
  | |   |                               |_ checksum
  | |   |_ code
  | |_ address
  |_ length
```

The linker has an option that controls the length of the output buffer for generating S1 records. The default buffer length is 32 code bytes.

The checksum calculation of S1 records is identical to S0.

S2 – record

With the linker option **–o***filename***:SREC:3**, the actual program code and data is supplied with S2 records, with the following layout:

’S’ ’2’ <*length_byte*> <*address*> <*code bytes*> <*checksum_byte*>

This record is used for 3–byte addresses.

Example:

```
S213FF002000232222754E00754F04AF4FAE4E22BF
  | |     |                            |_ checksum
  | |     |_ code
  | |_ address
  |_ length
```

The linker has an option that controls the length of the output buffer for generating S2 records. The default buffer length is 32 code bytes.

The checksum calculation of S2 records is identical to S0.

S3 – record

With the linker option **–o***filename***:SREC:4**, which is the default, the actual program code and data is supplied with S3 records, with the following layout:

’S’ ’3’ <*length_byte*> <*address*> <*code bytes*> <*checksum_byte*>

The linker generates 4–byte addresses by default.

Example:

```
S3070000FFFE6E6825
  | |       |   |_ checksum
  | |       |_ code
  | |_ address
  |_ length
```

The linker has an option that controls the length of the output buffer for generating S3 records.

The checksum calculation of S3 records is identical to S0.

S7 – record

With the linker option **–o***filename***:SREC:4**, which is the default, at the end of an S–record file, the linker generates an S7 record, which contains the program start address. S7 is the corresponding termination record for S3 records.

Layout:

’S’ ’7’ <*length_byte*> <*address*> <*checksum_byte*>

Example:

```
S70500000000FA
  | |       |_checksum
  | |_ address
  |_ length
```

The checksum calculation of S7 records is identical to S0.

S8 – record

With the linker option **–o***filename***:SREC:3**, at the end of an S–record file, the linker generates an S8 record, which contains the program start address.

Layout:

’S’ ’8’ <*length_byte*> <*address*> <*checksum_byte*>

Example:

```
S804FF0003F9
  | |     |_checksum
  | |_ address
  |_ length
```

The checksum calculation of S8 records is identical to S0.

S9 – record

With the linker option **–o***filename***:SREC:2**, at the end of an S–record file, the linker generates an S9 record, which contains the program start address. S9 is the corresponding termination record for S1 records.

Layout:

’S’ ’9’ <*length_byte*> <*address*> <*checksum_byte*>

Example:

```
S9030210EA
  | |   |_checksum
  | |_ address
  |_ length
```

The checksum calculation of S9 records is identical to S0.

6.3 Intel Hex Record Format

Intel Hex records describe the hexadecimal object file format for 8–bit, 16–bit and 32–bit microprocessors. The hexadecimal object file is an ASCII representation of an absolute binary object file. There are six different types of records:

- Data Record (8–, 16, or 32–bit formats)
- End of File Record (8–, 16, or 32–bit formats)
- Extended Segment Address Record (16, or 32–bit formats)
- Start Segment Address Record (16, or 32–bit formats)
- Extended Linear Address Record (32–bit format only)
- Start Linear Address Record (32–bit format only)

By default the linker generates records in the 32–bit format (4–byte addresses).

General Record Format

In the output file, the record format is:

:	*length*	*offset*	*type*	*content*	*checksum*

Where:

:	is the record header.
length	is the record length which specifies the number of bytes of the *content* field. This value occupies one byte (two hexadecimal digits). The linker outputs records of 255 bytes (32 hexadecimal digits) or less; that is, *length* is never greater than FFH.
offset	is the starting load offset specifying an absolute address in memory where the data is to be located when loaded by a tool. This field is two bytes long. This field is only used for Data Records. In other records this field is coded as four ASCII zero characters ('0000').
type	is the record type. This value occupies one byte (two hexadecimal digits). The record types are:

Byte Type	Record type
00	Data
01	End of File
02	Extended segment address (not used)
03	Start segment address (not used)
04	Extended linear address (32–bit)
05	Start linear address (32–bit)

content	is the information contained in the record. This depends on the record type.
checksum	is the record checksum. The linker computes the checksum by first adding the binary representation of the previous bytes (from *length* to *content*). The linker then computes the result of sum modulo 256 and subtracts the remainder from 256 (two's complement). Therefore, the sum of all bytes following the header is zero.

Extended Linear Address Record

The Extended Linear Address Record specifies the two most significant bytes (bits 16−31) of the absolute address of the first data byte in a subsequent Data Record:

:	02	0000	04	*upper_address*	*checksum*

The 32−bit absolute address of a byte in a Data Record is calculated as:

(*address* + *offset* + *index*) modulo 4G

where:

address	is the base address, where the two most significant bytes are the *upper_address* and the two least significant bytes are zero.
offset	is the 16−bit offset from the Data Record.
index	is the index of the data byte within the Data Record (0 for the first byte).

Example:

```
:0200000400FFFB
 | |   | |   |_ checksum
 | |   | |_ upper_address
 | |   |_ type
 | |_ offset
 |_ length
```

Data Record

The Data Record specifies the actual program code and data.

:	*length*	*offset*	00	*data*	*checksum*

The *length* byte specifies the number of *data* bytes. The linker has an option that controls the length of the output buffer for generating Data records. The default buffer length is 32 bytes.

The *offset* is the 16−bit starting load offset. Together with the address specified in the Extended Address Record it specifies an absolute address in memory where the data is to be located when loaded by a tool.

Example:

```
:0F002000002322222754E00754F04AF4FAE4E22C3
 | |   | |                             |_ checksum
 | |   | |_ data
 | |   |_ type
 | |_ offset
 |_ length
```

Start Linear Address Record

The Start Linear Address Record contains the 32-bit program execution start address.

Layout:

:	04	0000	05	*address*	*checksum*

Example:

```
:0400000500FF0003F5
 | |   | |       |_ checksum
 | |   | |_ address
 | |   |_ type
 | |_ offset
 |_ length
```

End of File Record

The hexadecimal file always ends with the following end-of-file record:

```
:00000001FF
 | |   | |_ checksum
 | |   |_ type
 | |_ offset
 |_ length
```

7 Linker Script Language

Summary

This chapter describes the syntax of the linker script language (LSL)

7.1 Introduction

To make full use of the linker, you can write a script with information about the architecture of the target processor and locating information. The language for the script is called the *Linker Script Language* (LSL). This chapter first describes the structure of an LSL file. The next section contains a summary of the LSL syntax. Finally, in the remaining sections, the semantics of the Linker Script Language is explained.

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

LSL serves two purposes. First it enables you to specify the characteristics (that are of interest to the linker) of your specific target board and of the cores installed on the board. Second it enables you to specify how sections should be located in memory.

7.2 Structure of a Linker Script File

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

The architecture definition (required)

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

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

See section 7.5, *Semantics of the Architecture Definition* for detailed descriptions of LSL in the architecture definition.

The derivative definition

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

When you design an FPGA together with a PCB, the components on the FPGA become part of the board design and there is no need to distinguish between internal and external memory. For this reason you probably do not need to work with derivative definitions at all. There are, however, two situations where derivative definitions are useful:

1. When you re−use an FPGA design for several board designs it may be practical to write a derivative definition for the FPGA design and include it in the project LSL file.
2. 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 7.6, *Semantics of the Derivative Definition* for a detailed description of LSL in the derivative definition.

The processor definition

The *processor definition* describes an instance of a derivative. Typically the processor definition instantiates one derivative only (single−core processor). A processor that contains multiple cores having the same (homogeneous) or different (heterogeneous) architecture can also be described by instantiating multiple derivatives of the same or different types in separate processor definitions.

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.

See section 7.7, *Semantics of the Board Specification* for a detailed description of LSL in the processor definition.

The memory and bus definitions (optional)

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

See section 7.7.3, *Defining External Memory and Buses*, for more information on how to specify the external physical memory layout. *Internal* memory for a processor should be defined in the derivative definition for that processor.

The board specification

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

- convert a logical address to 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

The skeleton of a linker script file now looks as follows:

```
architecture architecture_name
{
    architecture definition
}

derivative derivative_name
{
    derivative definition
}

processor processor_name
{
    processor definition
}

memory definitions and/or bus definitions
```

```
section_layout space_name
{
    section placement statements
}
```

7.3 Syntax of the Linker Script Language

7.3.1 Preprocessing

When the linker loads an LSL file, the linker processes it with a C–style prepocessor. As such, it strips C and C++ comments. You can use the standard ISO C preprocessor directives, such as `#include`, `#define`, `#if`/`#else`/`#endif`.

For example:

```
#include "arch.lsl"
```

Preprocess and include the file `arch.lsl` at this point in the LSL file.

7.3.2 Lexical Syntax

The following lexicon is used to describe the syntax of the Linker Script Language:

A ::= *B*	= *A* is defined as *B*
A ::= *B C*	= *A* is defined as *B* and *C*; *B* is followed by *C*
A ::= *B* | *C*	= *A* is defined as *B* or *C*
<*B*>$^{0\|1}$	= zero or one occurrence of *B*
<*B*>$^{>=0}$	= zero of more occurrences of *B*
<*B*>$^{>=1}$	= one of more occurrences of *B*

IDENTIFIER	= a character sequence starting with 'a'–'z', 'A'–'Z' or '_'. Following characters may also be digits and dots '.'
STRING	= sequence of characters not starting with \n, \r or \t
DQSTRING	= " *STRING* " (double quoted string)
OCT_NUM	= octal number, starting with a zero (06, 045)
DEC_NUM	= decimal number, not starting with a zero (14, 1024)
HEX_NUM	= hexadecimal number, starting with '0x' (0x0023, 0xFF00)

OCT_NUM, *DEC_NUM* and *HEX_NUM* can be followed by a **k** (kilo), **M** (mega), or **G** (giga).

Characters in **bold** are characters that occur literally. Words in *italics* are higher order terms that are defined in the same or in one of the other sections.

To write comments in LSL file, you can use the C style '`/*    */`' or C++ style '`//`'.

7.3.3 Identifiers

```
arch_name          ::= IDENTIFIER
bus_name           ::= IDENTIFIER
core_name          ::= IDENTIFIER
derivative_name    ::= IDENTIFIER
file_name          ::= DQSTRING
group_name         ::= IDENTIFIER
mem_name           ::= IDENTIFIER
proc_name          ::= IDENTIFIER
section_name       ::= DQSTRING
space_name         ::= IDENTIFIER
stack_name         ::= section_name
symbol_name        ::= DQSTRING
```

7.3.4 Expressions

The expressions and operators in this section work the same as in 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
```

7.3.5 Built-in Functions

```
function_call       ::= absolute ( expr )
                      | addressof ( addr_id )
                      | exists ( section_name )
                      | max ( expr , expr )
                      | min ( expr , expr )
                      | sizeof ( size_id )

addr_id             ::= sect : section_name
                      | mem : mem_name
                      | group : group_name

size_id             ::= group : group_name
                      | mem : mem_name
                      | sect : section_name
```

- Every space, bus, memory, section or group your refer to, must be defined in the LSL file.
- The `addressof()` and `sizeof()` functions with the **`group`** or **`sect`** argument can only be used in the right hand side of an assignment. The `sizeof()` function with the **`mem`** argument can be used anywhere in section layouts.

You can use the following built-in functions in expressions. All functions return a numerical value. This value is a 64-bit signed integer.

absolute()

```
int absolute( expr )
```

Converts the value of *expr* to a positive integer.

```
absolute( ”labelA”-”labelB” )
```

addressof()

```
int addressof( addr_id )
```

Returns the address of *addr_id*, which is a named section, group or memory. To get the offset of the section with the name `asect`:

```
addressof( sect: ”asect”)
```

This function only works in assignments.

exists()

```
int exists( section_name )
```

The function returns 1 if the section *section_name* exists in one or more object file, 0 otherwise. If the section is not present in input object files, but generated from LSL, the result of this function is undefined.

To check whether the section `mysection` exists in one of the object files that is specified to the linker:

```
exists( ”mysection” )
```

max()

```
int max( expr, expr )
```

Returns the value of the expression that has the largest value. To get the highest value of two symbols:

```
max( ”sym1” , ”sym2”)
```

min()

```
int min( expr, expr )
```

Returns the value of the expression hat has the smallest value. To get the lowest value of two symbols:

```
min( ”sym1” , ”sym2”)
```

sizeof()

```
int sizeof( size_id )
```

Returns the size of the object (group, section or memory) the identifier refers to. To get the size of the section ”asection”:

```
sizeof( sect: ”asection” )
```

The **group** and **sect** arguments only works in assignments. The **mem** argument can be used anywhere in section layouts.

7.3.6 LSL Definitions in the Linker Script File

```
description         ::= <definition>>=1

definition           ::= architecture_definition
                      | derivative_definition
                      | board_spec
                      | section_definition
```

- At least one *architecture_definition* must be present in the LSL file.

7.3.7 Memory and Bus Definitions

```
mem_def             ::= memory mem_name { <mem_descr ;>>=0 }
```

- A *mem_def* defines a *memory* with the *mem_name* as a unique name.

```
mem_descr           ::= type = <reserved>0|1 mem_type
                      | mau = expr
                      | size = expr
                      | speed = number
                      | mapping
```

- A *mem_def* contains exactly one **type** statement.
- A *mem_def* contains exactly one **mau** statement (non–zero size).
- A *mem_def* contains exactly one **size** statement.
- A *mem_def* contains zero or one **speed** statement (default value is 1).
- A *mem_def* contains at least one *mapping*.

```
mem_type            ::= rom         // attrs = rx
                      | ram         // attrs = rw
                      | nvram       // attrs = rwx

bus_def             ::= bus bus_name { <bus_descr ;>>=0 }
```

- A *bus_def* statement defines a *bus* with the given *bus_name* as a unique name within a core architecture.

```
bus_descr          ::= mau = expr
                     | width = expr  // bus width, nr
                     |               // of data bits
                     | mapping       // legal destination
                                     // 'bus' only
```

- The **`mau`** and **`width`** statements appear exactly once in a *`bus_descr`*. The default value for **`width`** is the **`mau`** size.
- The bus width must be an integer times the bus MAU size.
- The MAU size must be non-zero.
- A bus can only have a *`mapping`* on a destination *bus* (through **`dest = bus:`**).

```
mapping            ::= map ( map_descr <, map_descr>>=0 )

map_descr          ::= dest = destination
                     | dest_dbits = range
                     | dest_offset = expr
                     | size = expr
                     | src_dbits = range
                     | src_offset = expr
```

- A *`mapping`* requires at least the **`size`** and **`dest`** statements.
- Each *`map_descr`* can occur only once.
- You can define multiple mappings from a single source.
- Overlap between source ranges or destination ranges is not allowed.
- If the **`src_dbits`** or **`dest_dbits`** statement is not present, its value defaults to the **`width`** value if the source/destination is a bus, and to the **`mau`** size otherwise.

```
destination        ::= space : space_name
                     | bus : <proc_name |
                             core_name :>0|1 bus_name
```

- A *`space_name`* refers to a defined address space.
- A *`proc_name`* refers to a defined processor.
- A *`core_name`* refers to a defined core.
- A *`bus_name`* refers to a defined bus.
- The following mappings are allowed (source to destination)
  - space => space
  - space => bus
  - bus => bus
  - memory => bus

```
range              ::= number .. number
```

7.3.8 Architecture Definition

```
architecture_definition
                  ::= architecture arch_name
                      <( parameter_list )>0|1
                      <extends arch_name
                              <( argument_list )>0|1 >0|1
                      { arch_spec>=0 }
```

- An *architecture_definition* defines a core *architecture* with the given *arch_name* as a unique name.
- At least one *space_def* and at least one *bus_def* have to be present in an *architecture_definition*.
- An *architecture_definition* that uses the **extends** construct defines an architecture that *inherits* all elements of the architecture defined by the second *arch_name*. The *parent architecture* must be defined in the LSL file as well.

```
parameter_list    ::= parameter <, parameter>>=0

parameter         ::= IDENTIFIER <= expr>0|1

argument_list     ::= expr <, expr>>=0

arch_spec         ::= bus_def
                    | space_def
                    | endianness_def

space_def         ::= space space_name { <space_descr;>>=0 }
```

- A *space_def* defines an address space with the given *space_name* as a unique name within an architecture.

```
space_descr       ::= space_property ;
                    | section_definition  //no space ref

space_property    ::= id = number // as used in object
                    | mau = expr
                    | align = expr
                    | page_size = expr
                    | stack_def
                    | heap_def
                    | copy_table_def
                    | start_address
                    | mapping
```

- A *space_def* contains exactly one **id** and one **mau** statement.
- A *space_def* contains at most one **align** statement.
- A *space_def* contains at most one **page_size** statement.
- A *space_def* contains at least one mapping.

```
stack_def          ::= stack stack_name ( stack_heap_descr
                                           <, stack_heap_descr >>=0 )
```

- A *stack_def* defines a stack with the *stack_name* as a unique name.

```
heap_def           ::= heap heap_name ( stack_heap_descr
                                         <, stack_heap_descr >>=0 )
```

- A *heap_def* defines a heap with the *heap_name* as a unique name.

```
copy_table_def     ::= copytable ( copy_table_descr
                                    <, copy_table_descr>>=0 )
```

- A *space_def* contains at most one **copytable** statement.
- If the architecture definition contains more than one address space, exactly one copy table must be defined in one of the spaces. If the the architecture definition contains only one address space, a copy table definition is optional (it will be generated in the space).

```
stack_heap_descr   ::= min_size = expr
                     | grows = direction
                     | align = expr
                     | fixed
```

- The **min_size** statement must be present.
- You can specify at most one **align** statement and one **grows** statement.

```
direction          ::= low_to_high
                     | high_to_low
```

- If you do not specify the **grows** statement, the stack and grow **low-to-high**.

```
copy_table_descr   ::= align = expr
                     | copy_unit = expr
                     | dest <space_name>0|1 = space_name
```

- The **copy_unit** is defined by the size in MAUs in which the startup code moves data.
- The **dest** statement is only required when the startup code initializes memory used by another processor that has no access to ROM.
- A *space_name* refers to a defined address space.

```
start_addr         ::= start_address ( start_addr_descr
                                        <, start_addr_descr>>=0 )

start_addr_descr   ::= run_addr = expr
                     | symbol = symbol_name
```

- A *symbol_name* refers to the section that contains the startup code.

```
endianness_def     ::= endianness { <endianness_type;>>=1 }

endianness_type    ::= big
                     | little
```

7.3.9 Derivative Definition

```
derivative_definition
                  ::= derivative derivative_name
                      <( parameter_list )>0|1
                      <extends derivative_name
                              <( argument_list )>0|1 >0|1
                      { <derivative_spec>>=0 }
```

- A *derivative_definition* defines a derivative with the given *derivative_name* as a unique name.
- At least one *core_def* must be present in a *derivative_definition*.

```
derivative_spec   ::= core_def
                    | bus_def
                    | mem_def
                    | section_definition // no processor name

core_def          ::= core core_name { <core_descr ;>>=0 }
```

- A *core_def* defines a *core* with the given *core_name* as a unique name.

```
core_descr        ::= architecture = arch_name
                      <( argument_list )>0|1
                    | endianness = ( endianness_type
                               <, endianness_type>>=0 )
```

- An *arch_name* refers to a defined core architecture.
- Exactly one **architecture** statement must be present in a *core_def*.

7.3.10 Processor Definition and Board Specification

```
board_spec        ::= proc_def
                    | bus_def
                    | mem_def

proc_def          ::= processor proc_name
                      { proc_descr ; }

proc_descr        ::= derivative = derivative_name
                      <( argument_list )>0|1
```

- A *proc_def* defines a *processor* with the *proc_name* as a unique name.
- If you do not explicitly define a processor for a derivative in an LSL file, the linker defines a processor with the same name as that derivative.
- A *derivative_name* refers to a defined derivative.
- A *proc_def* contains exactly one **derivative** statement.

7.3.11 Section Layout Definition

```
section_definition ::= section_layout <space_ref>0|1
                        <( locate_direction )>0|1
                        { <section_statement>>=0 }
```

- A section definition inside a space definition does not have a *space_ref*.
- All global section definitions have a *space_ref*.

```
space_ref           ::= <proc_name>0|1 : <core_name>0|1
                          : space_name
```

- If more than one processor is present, the *proc_name* must be given for a global section layout.
- If the section layout refers to a processor that has more than one core, the *core_name* must be given in the *space_ref*.
- A *proc_name* refers to a defined processor.
- A *core_name* refers to a defined core.
- A *space_name* refers to a defined address space.

```
locate_direction    ::= direction = direction

direction           ::= low_to_high
                      | high_to_low
```

- A section layout contains at most one **direction** statement.
- If you do not specify the **direction** statement, the locate direction of the section layout is **low-to-high**.

```
section_statement
                    ::= simple_section_statement ;
                      | aggregate_section_statement

simple_section_statement
                    ::= assignment
                      | select_section_statement
                      | special_section_statement

assignment          ::= symbol_name assign_op expr

assign_op           ::= =
                      | :=

select_section_statement
                    ::= select <section_name>0|1
                        <section_selections>0|1
```

- Either a *section_name* or at least one *section_selection* must be defined.

```
section_selections
                    ::= ( section_selection
                          <, section_selection>>=0 )
```

```
section_selection
                    ::= attributes = < <+|-> attribute>>0
```

- *+attribute* means: select all sections that have this attribute.
- *-attribute* means: select all sections that do not have this attribute.

```
special_section_statement
                    ::= heap stack_name <size_spec>0|1
                      | stack stack_name <size_spec>0|1
                      | copytable
                      | reserved <section_name>0|1 <reserved_specs>0|1
```

- Special sections cannot be selected in load-time groups.

```
size_spec           ::= ( size = expr )

reserved_specs      ::= ( reserved_spec <, reserved_spec>>=0 )

reserved_spec       ::= attributes
                      | fill_spec
                      | size = expr
                      | alloc_allowed = absolute
```

- If a **reserved** section has attributes **r**, **rw**, **x**, **rx** or **rwx**, and no fill pattern is defined, the section is filled with zeros. If no attributes are set, the section is created as a scratch section (attributes **ws**, no image).

```
aggregate_section_statement
                    ::= { <section_statement>>=0 }
                      | group_descr
                      | if_statement
                      | section_creation_statement

group_descr         ::= group <group_name>0|1 <( group_specs )>0|1
                              section_statement
```

- No two groups for an address space can have the same *group_name*.

```
group_specs         ::= group_spec <, group_spec >>=0

group_spec          ::= group_alignment
                      | attributes
                      | group_load_address
                      | fill <= fill_values>0|1
                      | group_page
                      | group_run_address
                      | group_type
                      | allow_cross_references
```

- The **allow-cross-references** property is only allowed for *overlay* groups.
- Sub groups inherit all properties from a parent group.

```
group_alignment     ::= align = expr
```

```
attributes          ::= attributes = <attribute>>=1

group_load_address
                    ::= load_addr <= load_or_run_addr>0|1

fill_spec           ::= fill = fill_values

fill_values         ::= expr
                      | [ expr <, expr>>=0 ]

group_page          ::= page <= expr>0|1

group_run_address   ::= run_addr <= load_or_run_addr>0|1

group_type          ::= clustered
                      | contiguous
                      | ordered
                      | overlay
```

- For *non-contiguous* groups, you can only specify `group_alignment` and `attributes`.
- The **`overlay`** keyword also sets the **`contiguous`** property.
- The **`clustered`** property cannot be set together with **`contiguous`** or **`ordered`** on a single group.

```
attribute           ::= r    // readable sections
                      | w    // writable sections
                      | x    // executable code sections
                      | i    // initialized sections
                      | s    // scratch sections
                      | b    // blanked (cleared) sections

load_or_run_addr    ::= addr_absolute
                      | addr_range <| addr_range>>=0

addr_absolute       ::= expr
                      | memory_reference [ expr ]
```

- An absolute address can only be set on *ordered* groups.

```
addr_range          ::= [ expr .. expr ]
                      | memory_reference
                      | memory_reference [ expr .. expr ]
```

- The parent of a group with an `addr_range` or **`page`** restriction cannot be **`ordered`**, **`contiguous`** or **`clustered`**.

```
memory_reference    ::= mem : <proc_name :>0|1 <core_name :>0|1 mem_name
```

- A `proc_name` refers to a defined processor.
- A `core_name` refers to a defined core.
- A `mem_name` refers to a defined memory.

```
if_statement        ::= if ( expr ) section_statement
                        <else section_statement>0|1
```

```
section_creation_statement
                ::= section section_name ( <section_spec>0|1 )
                    { <select_section_statement ;>>=0 }

section_spec    ::= attributes
                  | fill_spec
                  | size = expr
```

7.4 Expression Evaluation

Only *constant* expressions are allowed, including sizes, but not addresses, of sections in object files.

All expressions are evaluated with 64−bit precision integer arithmetic. The result of an expression can be absolute or relocatable. A symbol you assign is created as an absolute symbol.

7.5 Semantics of the Architecture Definition

Keywords in the architecture definition

```
architecture
   extends
endianness          big  little
bus
   mau
   width
   map
space
   id
   mau
   align
   page_size
   stack
      min_size
      grows         low_to_high  high_to_low
      align
      fixed
   heap
      min_size
      grows         low_to_high  high_to_low
      align
      fixed
   copytable
      align
      copy_unit
      dest
   start_address
      run_addr
      symbol
   map

   map
      dest          bus  space
      dest_dbits
      dest_offset
      size
      src_dbits
      src_offset
```

7.5.1 Defining an Architecture

With the keyword **architecture** you define an architecture and assign a unique name to it. The name is used to refer to it at other places in the LSL file:

```
architecture name
{
      definitions
}
```

If you are defining multiple core architectures that show great resemblance, you can define the common features in a parent core architecture and extend this with a child core architecture that contains specific features. The child inherits all features of the parent. With the keyword **extends** you create a child core architecture:

```
architecture name_child_arch extends name_parent_arch
{
      definitions
}
```

A core architecture can have any number of parameters. These are identifiers which get values assigned on instantiation or extension of the architecture. You can use them in any expression within the core architecture. Parameters can have default values, which are used when the core architecture is instantiated with less arguments than there are parameters defined for it. When you extend a core architecture you can pass arguments to the parent architecture. Arguments are expressions that set the value of the parameters of the sub–architecture.

```
architecture name_child_arch (parm1,parm2=1)
             extends name_parent_arch (arguments)
{
      definitions
}
```

7.5.2 Defining Internal Buses

With the **bus** keyword you define a bus (the combination of data and corresponding address bus). The bus name is used to identify a bus and does not conflict with other identifiers. Bus descriptions in an architecture definition or derivative definition define *internal* buses. Some internal buses are used to communicate with the components outside the core or processor. Such buses on a processor have physical pins reserved for the number of bits specified with the **width** statements.

- The **mau** field specifies the MAU size (Minimum Addressable Unit) of the data bus. This field is required.
- The **width** field specifies the width (number of address lines) of the data bus. The default value is the MAU size.
- The **map** keyword specifies how this bus maps onto another bus (if so). Mappings are described in section 7.5.4, *Mappings*.

```
bus bus_name
{
      mau = 8;
      width = 8;
      map ( map_description );
}
```

7.5.3 Defining Address Spaces

With the **space** keyword you define a logical address space. The space name is used to identify the address space and does not conflict with other identifiers.

- The **id** field defines how the addressing space is identified in object files. In general, each address space has a unique ID. The linker locates sections with a certain ID in the address space with the same ID. This field is required. In IEEE this ID is specified explicitly for sections and symbols, ELF sections map by default to the address space with ID 1. Sections with one of the special names defined in the ABI (Application Binary Interface) may map to different address spaces.
- The **mau** field specifies the MAU size (Minimum Addressable Unit) of the space. This field is required.
- The **align** value must be a power of two. The linker uses this value to compute the start addresses when sections are concatenated. An align value of *n* means that objects in the address space have to be aligned on *n* MAUs.
- The **page_size** field sets the page size in MAUs for the address space. It must be a power of 2. The default page size is 1. See also the **page** keyword in subsection *Locating a group* in section 7.8.2, *Creating and Locating Groups of Sections*.
- The **map** keyword specifies how this address space maps onto an internal bus or onto another address space. Mappings are described in section 7.5.4, *Mappings*.

Stacks and heaps

- The **stack** keyword defines a stack in the address space and assigns a name to it. The architecture definition must contain at least one stack definition. Each stack of a core architecture must have a unique name. See also the **stack** keyword in section 7.8.3, *Creating or Modifying Special Sections*.

  The stack is described in terms of a minimum size (**min_size**) and the direction in which the stack grows (**grows**). This can be either from **low_to_high** addresses (stack grows upwards, this is the default) or from **high_to_low** addresses (stack grows downwards). The **min_size** is required.

  By default, the linker tries to maximize the size of the stacks and heaps. After locating all sections, the largest remaining gap in the space is used completely for the stacks and heaps. If you specify the keyword **fixed**, you can disable this so-called ’balloon behavior’. The size is also fixed if you used a stack or heap in the software layout definition in a restricted way. For example when you override a stack with another size or select a stack in an ordered group with other sections.

  Optionally you can specify an alignment for the stack with the argument **align**. This alignment must be equal or larger than the alignment that you specify for the address space itself.

- The **heap** keyword defines a heap in the address space and assigns a name to it. The definition of a heap is similar to the definition of a stack. See also the **heap** keyword in section 7.8.3, *Creating or Modifying Special Sections*.

See section 7.8, *Semantics of the Section Layout Definition* for information on creating and placing stack sections.

Copy tables

- The **copytable** keyword defines a copy table in the address space. The content of the copy table is created by the linker and contains the start address and size of all sections that should be initialized by the startup code. If the architecture definition contains more than one address space, you must define exactly one copy table in one of the address spaces. If the architecture definition contains only one address space, the copy table definition is optional.

  Optionally you can specify an alignment for the copy table with the argument **align**. This alignment must be equal or larger than the alignment that you specify for the address space itself. If smaller, the alignment for the address space is used.

  The **copy_unit** argument specifies the size in MAUs of information chunks that are copied. If you do not specify the copy unit, the MAU size of the address space itself is used.

  The **dest** argument specifies the destination address space that the code uses for the copy table. The linker uses this information to generate the correct addresses in the copy table. The memory into where the sections must be copied at run–time, must be accessible from this destination space.

Start address

- The **start_address** keyword specifies the start address for the position where the C startup code is located. When a processor is reset, it initializes its program counter to a certain start address, sometimes called the *reset vector*. In the architecture definition, you must specify this start address in the correct address space in combination with the name of the label in the application code which must be located here.

  The **run_addr** argument specifies the start address (reset vector). If the core starts executing using an entry from a vector table, and directly jumps to the start label, you should omit this argument.

  The **symbol** argument specifies the name of the label in the application code that should be located at the specified start address. The **symbol** argument is required. The linker will resolve the start symbol and use its value after locating for the start address field in IEEE–695 files and Intel Hex files. If you also specified the **run_addr** argument, the start symbol (label) must point to a section. The linker locates this section such that the start symbol ends up on the start address.

```
space space_name
{
      id = 1;
      mau = 8;
      align = 8;
      page_size = 1;
      stack name (min_size = 1k, grows = low_to_high);
      start_address ( run_addr = 0x0000,
                      symbol = "start_label" )
      map ( map_description );
}
```

7.5.4 Mappings

You can use a mapping when you define a space, bus or memory. With the **map** field you specify how addresses from the source (space, bus or memory) are translated to addresses of a destination (space, bus). The following mappings are possible:

- space => space
- space => bus
- bus => bus
- memory => bus

With a mapping you specify a range of source addresses you want to map (specified by a source offset and a size), the destination to which you want to map them (a bus or another address space), and the offset address in the destination.

- The **dest** argument specifies the destination. This can be a **bus** or another address **space** (only for a space to space mapping). This argument is required.
- The **src_offset** argument specifies the offset of the source addresses. In combination with size, this specifies the range of address that are mapped. 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).

From space to space

If you map an address space to another address space (nesting), you can do this by mapping the subspace to the containing larger space. In this example a small space of 64k is mapped on a large space of 16M.

```
space small
{
   id = 2;
   mau = 4;
   map (src_offset = 0, dest_offset = 0,
        dest = space : large, size = 64k);
}
```

From space to bus

All spaces that are not mapped to another space must map to a bus in the architecture:

```
space large
{
   id = 1;
   mau = 4;
   map (src_offset = 0, dest_offset = 0,
        dest = bus:bus_name, size = 16M );
}
```

From bus to bus

The next example maps an external bus called `e_bus` to an internal bus called `i_bus`. This internal bus resides on a core called `mycore`. The source bus has 16 data lines whereas the destination bus has only 8 data lines. Therefore, the keywords **`src_dbits`** and **`dest_dbits`** specify which source data lines are mapped on which destination data lines.

```
architecture mycore
{
    bus i_bus
    {
       mau = 4;
    }

    space i_space
    {
       map (dest=bus:i_bus, size=256);
    }
}
```

```
bus e_bus
{
   mau = 16;
   width = 16;
   map (dest = bus:mycore:i_bus, src_dbits = 0..7, dest_dbits = 0..7 )
}
```

It is not possible to map an internal bus to an external bus.

7.6 Semantics of the Derivative Definition

Keywords in the derivative definition

```
derivative
   extends
core
   architecture
bus
   mau
   width
   map
memory
   type             reserved  rom  ram  nvram
   mau
   size
   speed
   map

   map
      dest          bus  space
      dest_dbits
      dest_offset
      size
      src_dbits
      src_offset
```

7.6.1 Defining a Derivative

With the keyword **`derivative`** you define a derivative and assign a unique name to it. The name is used to refer to it at other places in the LSL file:

```
derivative name
{
     definitions
}
```

If you are defining multiple derivatives that show great resemblance, you can define the common features in a parent derivative and extend this with a child derivative that contains specific features. The child inherits all features of the parent (cores and memories). With the keyword **`extends`** you create a child derivative:

```
derivative name_child_deriv extends name_parent_deriv
{
     definitions
}
```

As with a core architecture, a derivative can have any number of parameters. These are identifiers which get values assigned on instantiation or extension of the derivative. You can use them in any expression within the derivative definition.

```
derivative name_child_deriv (parm1,parm2=1)
           extends name_parent_derivh (arguments)
{
     definitions
}
```

7.6.2 Instantiating Core Architectures

With the keyword **`core`** you instantiate a core architecture in a derivative.

- With the keyword **`architecture`** you tell the linker that the given core has a certain architecture. The architecture name refers to an existing architecture definition in the same LSL file.

  For example, if you have two cores (called `mycore_1` and `mycore_2`) that have the same architecture (called `mycorearch`), you must instantiate both cores as follows:

```
core mycore_1
{
     architecture = mycorearch;
}

core mycore_2
{
     architecture = mycorearch;
}
```

  If the architecture definition has parameters you must specify the arguments that correspond with the parameters. For example `mycorearch1` expects two parameters which are used in the architecture definition:

```
core mycore
{
     architecture = mycorearch1 (1,2);
}
```

7.6.3 Defining Internal Memory and Buses

With the **`memory`** keyword you define physical memory that is present on the target board. The memory name is used to identify the memory and does not conflict with other identifiers. It is common to define internal memory (on–chip) in the derivative definition. External memory (off–chip memory) is usually defined in the board specification (See section 7.7.3, *Defining External Memory and Buses*).

- The **`type`** field specifies a memory type:
  - **`rom`**: read only memory
  - **`ram`**: random access memory
  - **`nvram`**: non volatile ram

The optional **reserved** qualifier before the memory type, tells the linker not to locate any section in the memory by default. You can locate sections in such memories using an absolute address or range restriction (see subsection *Locating a group* in section 7.8.2, *Creating and Locating Groups of Sections*).

- The **mau** field specifies the MAU size (Minimum Addressable Unit) of the memory. This field is required.
- The **size** field specifies the size in MAU of the memory. This field is required.
- The **speed** field specifies a symbolic speed for the memory (0..4): 0 is the fastest, 4 the slowest. The linker uses the relative speed of the memories in such a way, that optimal speed is achieved. The default speed is 1.
- The **map** field specifies how this memory maps onto an (internal) bus. Mappings are described in section 7.5.4, *Mappings*.

```
memory mem_name
{
   type = rom;
   mau = 8;
   size = 64k;
   speed = 2;
   map ( map_description );
}
```

With the **bus** keyword you define a bus in a derivative definition. Buses are described in section 7.5.2, *Defining Internal Buses*.

7.7 Semantics of the Board Specification

Keywords in the board specification

```
processor
   derivative
bus
   mau
   width
   map
memory
   type                reserved  rom  ram  nvram
   mau
   size
   speed
   map

   map
      dest          bus  space
      dest_dbits
      dest_offset
      size
      src_dbits
      src_offset
```

7.7.1 Defining a Processor

If you have a target board with multiple processors that have the same derivative, you need to instantiate each individual processor in a processor definition. This information tells the linker which processor has which derivative and enables the linker to distinguish between the present processors.

If you use processors that all have a unique derivative, you may omit the processor definitions. In this case the linker assumes that for each derivative definition in the LSL file there is one processor. The linker uses the derivative name also for the processor.

With the keyword **`processor`** you define a processor. You can freely choose the processor name. The name is used to refer to it at other places in the LSL file:

```
processor proc_name
{
   processor definition
}
```

7.7.2 Instantiating Derivatives

With the keyword **derivative** you tell the linker that the given processor has a certain derivative. The derivative name refers to an existing derivative definition in the same LSL file.

For examples, if you have two processors on your target board (called `myproc_1` and `myproc_2`) that have the same derivative (called `myderiv`), you must instantiate both processors as follows:

```
processor myproc_1
{
     derivative = myderiv;
}

processor myproc_2
{
     derivative = myderiv;
}
```

If the derivative definition has parameters you must specify the arguments that correspond with the parameters. For example `myderiv1` expects two parameters which are used in the derivative definition:

```
processor myproc
{
     derivative = myderiv1 (2,4);
}
```

7.7.3 Defining External Memory and Buses

It is common to define external memory (off–chip) and external buses at the global scope (outside any enclosing definition). Internal memory (on–chip memory) is usually defined in the scope of a derivative definition.

With the keyword **memory** you define physical memory that is present on the target board. The memory name is used to identify the memory and does not conflict with other identifiers. If you define memory parts in the LSL file, only the memory defined in these parts is used for placing sections.

If no external memory is defined in the LSL file and if the linker option to allocate memory on demand is set then the linker will assume that all virtual addresses are mapped on physical memory. You can override this behavior by specifying one or more memory definitions.

```
memory mem_name
{
   type = rom;
   mau = 8;
   size = 64k;
   speed = 2;
   map ( map_description );
}
```

For a description of the keywords, see section 7.6.3, *Defining Internal Memory and Buses*.

With the keyword **bus** you define a bus (the combination of data and corresponding address bus). The bus name is used to identify a bus and does not conflict with other identifiers. Bus descriptions at the global scope (outside any definition) define *external* buses. These are buses that are present on the target board.

```
bus bus_name
{
      mau = 8;
      width = 8;
      map ( map_description );
}
```

For a description of the keywords, see section 7.5.2, *Defining Internal Buses*.

You can connect off-chip memory to any derivative: you need to map the off-chip memory to a bus and map that bus on the internal bus of the derivative you want to connect it to.

7.8 Semantics of the Section Layout Definition

Keywords in the section layout definition

```
section_layout
   direction      low_to_high  high_to_low
group
   align
   attributes     + -  r w x b i s
   fill
   ordered
   clustered
   contiguous
   overlay
   allow_cross_references
   load_addr
      mem
   run_addr
      mem
   page
select
heap
   size
stack
   size
reserved
   size
   attributes     r w x
   fill
   alloc_allowed absolute
copytable
section
   size
   attributes     r w x
   fill

if
else
```

7.8.1 Defining a Section Layout

With the keyword **`section_layout`** you define a section layout for exactly one address space. In the section layout you can specify how input sections are placed in the address space, relative to each other, and what the absolute run and load addresses of each section will be.

You can define one or more section definitions. Each section definition arranges the sections in one address space. You can precede the address space name with a processor name and/or core name, separated by colons. You can omit the processor name and/or the core name if only one processor is defined and/or only one core is present in the processor. A reference to a space in the only core of the only processor in the system would look like "`::my_space`". A reference to a space of the only core on a specific processor in the system could be "`my_chip::my_space`". The next example shows a section definition for sections in the `my_space` address space of the processor called `my_chip`:

```
section_layout my_chip::my_space ( locate_direction )
{
    section statements
}
```

With the optional keyword **`direction`** you specify whether the linker starts locating sections from **`low_to_high`** (default) or from **`high_to_low`**. In the second case the linker starts locating sections at the highest addresses in the address space but preserves the order of sections when necessary (one processor and core in this example).

```
section_layout ::my_space ( direction = high_to_low )
{
    section statements
}
```

If you do not explicitly tell the linker how to locate a section, the linker decides on the basis of the section attributes in the object file and the information in the architecture definition and memory parts where to locate the section.

7.8.2 Creating and Locating Groups of Sections

Sections are located per group. A group can contain one or more (sets of) input sections as well as other groups. Per group you can assign a mutual order to the sets of sections and locate them into a specific memory part.

```
group ( group_specifications )
{
    section_statements
}
```

With the *section_statements* you generally select sets of sections to form the group. This is described in subsection *Selecting sections for a group*.

Instead of selecting sections, you can also modify special sections like stack and heap or create a reserved section. This is described in section 7.8.3, *Creating or Modifying Special Sections*.

With the *group_specifications* you actually locate the sections in the group. This is described in subsection *Locating a group*.

Selecting sections for a group

With the **`select`** keyword you can select one or more sections for the group. You can select a section by name or by attributes. If you select a section by name, you can use a wildcard pattern:

"*"	matches with all section names
"?"	matches with a single character in the section name
"\"	takes the next character literally
"[abc]"	matches with a single 'a', 'b' or 'c' character
"[a–z]"	matches with any single character in the range 'a' to 'z'

```
group ( ... )
{
   select "mysection";
   select "*";
}
```

The first **`select`** statement selects the section with the name "mysection". The second **`select`** statement selects all sections that were not selected yet.

A section is selected by the first **`select`** statement that matches, in the union of all section layouts for the address space. Global section layouts are processed in the order in which they appear in the LSL file. Internal core architecture section layouts always take precedence over global section layouts.

- The **`attributes`** field selects all sections that carry (or do not carry) the given attribute. With +*attribute* you select sections that have the specified attribute set. With –*attribute* you select sections that do not have the specified attribute set. You can specify one or more of the following attributes:
  - **`r`** readable sections
  - **`w`** writable sections
  - **`x`** executable sections
  - **`i`** initialized sections
  - **`b`** sections that should be cleared at program startup
  - **`s`** scratch sections (not cleared and not initialized)

To select all read–only sections:

```
group ( ... )
{
   select (attributes = +r–w);
}
```

Keep in mind that all section selections are restricted to the address space of the section layout in which this group definition occurs.

Locating a group

```
group group_name ( group_specifications )
{
   section_statements
}
```

With the *group_specifications* you actually define how the linker must locate the group. You can roughly define three things: 1) assign properties to the group like alignment and read/write attributes, 2) define the mutual order in the address space for sections in the group and 3) restrict the possible addresses for the sections in a group.

The linker creates labels that allow you to refer to the begin and end address of a group from within the application software. Labels **`__lc_gb_`***`group_name`* and **`__lc_ge_`***`group_name`* mark the begin and end of the group respectively, where the begin is the lowest address used within this group and the end is the highest address used. Notice that a group not necessarily occupies all memory between begin and end address. The given label refers to where the section is located at run−time (versus load−time).

1. Assign properties to the group like alignment and read/write attributes.
   These properties are assigned to all sections in the group (and subgroups) and override the attributes of the input sections.
   - The **`align`** field tells the linker to align all sections in the group and the group as a whole according to the align value. By default the linker uses the largest alignment constraint of either the input sections or the alignment of the address space.
   - The **`attributes`** field tells the linker to assign one or more attributes to all sections in the group. This overrules the default attributes. By default the linker uses the attributes of the input sections. You can set the **`r`**, **`w`** or **`rw`** attributes and you can switch between the **`b`** and **`s`** attributes.
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 contigous range of memory. Due to alignment of sections there can be ’alignment gaps’ between the sections.

     When you define a group that is both **`ordered`** and **`contiguous`**, this is called a *sequential* group. In a sequential group the linker places sections in the same order in the address space as they appear in the group and it occupies a contiguous range of memory.

- The **`clustered`** keyword tells the linker to locate the sections in the group in a number of *contiguous* blocks. It tries to keep the number of these blocks to a minimum. If enough memory is available, the group will be located as if it was specified as **`contiguous`**. Otherwise, it gets split into two or more blocks.

  If a contiguous or clustered group contains *alignment gaps*, the linker can locate sections that are not part of the group in these gaps. To prevent this, you can use the **`fill`** keyword. If the group is located in RAM, the gaps are treated as reserved (scratch) space. If the group is located in ROM, the alignment gaps are filled with zeros by default. You can however change the fill pattern by specifying a bit pattern. The result of the expression, or list of expressions, is used as values to write to memory, each in MAU.

- The **`overlay`** keyword tells the linker to overlay the sections in the group. The linker places all sections in the address space using a contiguous range of addresses. (Thus an overlay group is automatically also a contiguous group.) To overlay the sections, all sections in the overlay group share the same run-time address.

  For each input section within the overlay the linker automatically defines two symbols. The symbol **`__lc_cb_`***`section_name`* is defined as the load-time start address of the section. The symbol **`__lc_ce_`***`section_name`* is defined as the load-time end address of the section. C (or assembly) code may be used to copy the overlaid sections.

  If sections in the overlay group contain references between groups, the linker reports an error. The keyword **`allow_cross_references`** tells the linker to accept cross-references. Normally, it does not make sense to have references between sections that are overlaid.

```
group ovl (overlay)
{
    group a
    {
        select "my_ovl_p1";
        select "my_ovl_p2";
    }
    group b
    {
        select "my_ovl_q1";
    }
}
```

It may be possible that one of the sections in the overlay group already has been defined in another group where it received a load-time address. In this case the linker does not overrule this load-time address and excludes the section from the overlay group.

3. Restrict the possible addresses for the sections in a group.
   The load−time address specifies where the group's elements are loaded in 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. With an *expression* you can specify that the group should be located at the absolute address specified by the expression:

     ```
     group (run_addr = 0xa00f0000)
     ```

     You can use the '`[`*`offset`*`]`' variant to locate the group at the given absolute offset in memory:

     ```
     group (run_addr = mem:A[0x1000])
     ```

     A range can be an absolute space address range, written as **`[`** *`expr`* **`..`** *`expr`* **`]`**, a complete memory device, written as **`mem:`***`mem_name`*, or a memory address range, **`mem:`***`mem_name`***`[`***`expr`* **`..`** *`expr`***`]`**

     ```
     group (run_addr = mem:my_dram)
     ```

     You can use the '`|`' to specify an address range of more than one physical memory device:

     ```
     group (run_addr = mem:A | mem:B)
     ```

   - The **`load_addr`** keyword changes the meaning of the section selection in the group: the linker selects the load−time ROM copy of the named section(s) instead of the regular sections. Just like **`run_addr`** you can specify an absolute address or an address range.

     The **`load_addr`** keyword itself (without an assignment) specifies that the group's position in the LSL file defines its load−time address.

     ```
     group (load_addr)
           select "mydata";  // select ROM copy of mydata: "[mydata]"
     ```

   The load−time and run−time addresses of a group cannot be set at the same time. If the load−time property is set for a group, the group (only) restricts the positioning at load−time of the group's sections. It is not possible to set the address of a group that has a not−unrestricted parent group.

   The properties of the load−time and run−time start address are:

   - At run−time, before using an element in an overlay group, the application copies the sections from their load location to their run−time location, but only if these two addresses are different. For non−overlay sections this happens at program start−up.
   - The start addresses cannot be set to absolute values for unrestricted groups.
   - For non−overlay groups that do not have an overlay parent, the load−time start address equals the run−time start address.
   - For any group, if the run−time start address is not set, the linker selects an appropriate address.

For overlays, the linker reserves memory at the run-time start address as large as the largest element in the overlay group.

- The **page** keyword tells the linker to place the group in one page. Instead of specifying a run-time address, you can specify a page and optional a page number. Page numbers start from zero. If you omit the page number, the linker chooses a page.

  The **page** keyword refers to pages in the address space as defined in the architecture definition. See also the **page_size** keyword in section 7.5.3, *Defining Address Spaces*.

  ```
  group (page, ... )
  group (page = 3, ...)
  ```

7.8.3 Creating or Modifying Special Sections

Instead of selecting sections, you can also create a reserved section or an output section or modify special sections like a stack or a heap. Because you cannot define these sections in the input files, you must use the linker to create them.

Stack

- The **stack** keyword tells the linker to reserve memory for the stack. The name for the stack section refers to the stack as defined in the architecture definition. If no name was specified in the architecture definition, the default name is `stack`.

  With the keyword **size** you can specify the size for the stack. If the **size** is not specified, the linker uses the size given by the **min_size** argument as defined for the stack in the architecture definition. Normally the linker automatically tries to maximize the size, unless you specified the **fixed** keyword.

  ```
  group ( ... )
  {
     stack "mystack" ( size = 2k );
  }
  ```

  The linker creates two labels to mark the begin and end of the stack, **__lc_ub_***stack_name* for the begin of the stack and **__lc_ue_***stack_name* for the end of the stack. The linker allocates space for the stack when there is a reference to either of the labels.

  See also the **stack** keyword in section 7.5.3, *Defining Address Spaces*.

Heap

- The **heap** keyword tells the linker to reserve a dynamic memory range for the `malloc()` function. Optionally you can assign a name to the heap section. With the keyword **size** you can change the size for the heap. If the **size** is not specified, the linker uses the size given by the **min_size** argument as defined for the heap in the architecture definition. Normally the linker automatically tries to maximize the size, unless you specified the **fixed** keyword.

```
group ( ... )
{
   heap ”myheap” ( size = 2k );
}
```

The linker creates two labels to mark the begin and end of the heap, **__lc_ub_***heap_name* for the begin of the heap and **__lc_ue_***heap_name* for the end of the heap. The linker allocates space for the heap when a reference to either of the section labels exists in one of the input object files.

Reserved section

- The **reserved** keyword tells the linker to create an area or section of a given size. The linker will not locate any other sections in the memory occupied by a reserved section, with some exceptions. Optionally you can assign a name to a reserved section. With the keyword **size** you can specify a size for a given reserved area or section.

```
group ( ... )
{
   reserved ”myreserved” ( size = 2k );
}
```

The optional **fill** field contains a bit pattern that the linker writes to all memory addresses that remain unoccupied during the locate process. The result of the expression, or list of expressions, is used as values to write to memory, each in MAU. The first MAU of the fill pattern is always the first MAU in the section.

By default, no sections can overlap with a reserved section. With **alloc_allowed=absolute** sections that are located at an absolute address due to an absolute group restriction can overlap a reserved section.

With the **attributes** field you can set the access type of the reserved section. The linker locates the reserved section in its space with the restrictions that follow from the used attributes, **r**, **w** or **x** or a valid combination of them. The allowed attributes are shown in the following table. A value between < and > in the table means this value is set automatically by the linker.

Properties set in LSL		Resulting section properties		
attributes	**filled**	**access**	**memory**	**content**
x	yes		<rom>	executable
r	yes	r	<rom>	data
r	no	r	<rom>	scratch
rx	yes	r	<rom>	executable
rw	yes	rw	<ram>	data
rw	no	rw	<ram>	scratch
rwx	yes	rw	<ram>	executable

```
group ( ... )
{
   reserved ”myreserved” ( size = 2k,
                attributes = rw, fill = 0xaa );
}
```

If you do not specify any attributes, the linker will reserve the given number of maus, no matter what type of memory lies beneath. If you do not specify a fill pattern, no section is generated.

The linker creates two labels to mark the begin and end of the section, **__lc_ub_***name* for the start, and **__lc_ue_***name* for the end of the reserved section.

Output sections

- The **section** keyword tells the linker to accumulate sections obtained from object files (”input sections”) into an output section of a fixed size in the locate phase. You can select the input sections with **select** statements. With the keyword **size** you specify the size of the output section.

  The **fill** field contains a bit pattern that the linker writes to all unused space in the output section. When all input sections have an image (code/data) you must specify a fill pattern. If you do not specify a fill pattern, all input sections must be scratch sections. The fill pattern is aligned at the start of the output section.

  As with a reserved section you can use the **attributes** field to set the access type of the output section.

```
group ( ... )
{
   section ”myoutput” ( size = 4k, attributes = rw, fill = 0xaa )
   {
      select ”myinput1”;
      select ”myinput2”;
   }
}
```

The linker creates two labels to mark the begin and end of the section, **__lc_ub_***name* for the start, and **__lc_ue_***name* for the end of the output section.

Copy table

- The **copytable** keyword tells the linker to select a section that is used as *copy−table*. The content of the copy−table is created by the linker. It contains the start address and length of all sections that should be initialized by the startup code.

  The linker creates two labels to mark the begin and end of the section, **__lc_ub_table** for the start, and **__lc_ue_table** for the end of the copy table. The linker generates a copy table when a reference to either of the section labels exists in one of the input object files.

7.8.4 Creating Symbols

You can tell the linker to create symbols before locating by putting assignments in the section layout definition. Symbol names are represented by double−quoted strings. Any string is allowed, but object files may not support all characters for symbol names. You can use two different assignment operators. With the simple assignment operator '**=**', the symbol is created unconditionally. With the '**:=**' operator, the symbol is only created if it already exists as an undefined reference in an object file.

The expression that represents the value to assign to the symbol may contain references to other symbols. If such a referred symbol is a special section symbol, creation of the symbol in the left hand side of the assignment will cause creation of the special section.

```
section_layout
{
   "__lc_bs" := "__lc_ub_stack";
    // when the symbol __lc_bs occurs as an undefined reference
    // in an object file, the linker allocates space for the stack
}
```

7.8.5 Conditional Group Statements

Within a group, you can conditionally select sections or create special sections.

- With the **`if`** keyword you can specify a condition. The succeeding section statement is executed if the condition evaluates to TRUE (1).
- The optional **`else`** keyword is followed by a section statement which is executed in case the if−condition evaluates to FALSE (0).

```
group ( ... )
{
   if ( exists ( "mysection" ) )
      select "mysection";
   else
      reserved "myreserved" ( size=2k );
}
```

8 MISRA C Rules

Summary

This chapter contains an overview of the supported and unsupported MISRA C rules.

Supported and unsupported MISRA C rules

A number of MISRA C rules leave room for interpretation. Other rules can only be checked in a limited way. In such cases the implementation decisions and possible restrictions for these rules are listed.

x means that the rule is not supported by the TASKING C compiler.

(R) is a required rule, (A) is an advisory rule.

	1.	(R)	The code shall conform to standard C, without language extensions
x	2.	(A)	Other languages should only be used with an interface standard
	3.	(A)	Inline assembly is only allowed in dedicated C functions
x	4.	(A)	Provision should be made for appropriate run–time checking
	5.	(R)	Only use characters and escape sequences defined by ISO C
x	6.	(R)	Character values shall be restricted to a subset of ISO 106460–1
	7.	(R)	Trigraphs shall not be used
	8.	(R)	Multibyte characters and wide string literals shall not be used
	9.	(R)	Comments shall not be nested
	10.	(A)	Sections of code should not be ”commented out” In general, it is not possible to decide whether a piece of comment is C code that is commented out, or just some pseudo code. Instead, the following heuristics are used to detect possible C code inside a comment: – a line ends with ’;’, or – a line starts with ’}’, possibly preceded by white space
	11.	(R)	Identifiers shall not rely on significance of more than 31 characters
	12.	(A)	The same identifier shall not be used in multiple name spaces
	13.	(A)	Specific–length typedefs should be used instead of the basic types
	14.	(R)	Use ’unsigned char’ or ’signed char’ instead of plain ’char’
x	15.	(A)	Floating–point implementations should comply with a standard

16. (R) The bit representation of floating−point numbers shall not be used

    A violation is reported when a pointer to a floating−point type is converted to a pointer to an integer type.

17. (R) ”typedef” names shall not be reused
18. (A) Numeric constants should be suffixed to indicate type

    A violation is reported when the value of the constant is outside the range indicated by the suffixes, if any.

19. (R) Octal constants (other than zero) shall not be used
20. (R) All object and function identifiers shall be declared before use
21. (R) Identifiers shall not hide identifiers in an outer scope
22. (A) Declarations should be at function scope where possible
23. x (A) All declarations at file scope should be static where possible
24. (R) Identifiers shall not have both internal and external linkage
25. x (R) Identifiers with external linkage shall have exactly one definition
26. (R) Multiple declarations for objects or functions shall be compatible
27. x (A) External objects should not be declared in more than one file
28. (A) The ”register” storage class specifier should not be used
29. (R) The use of a tag shall agree with its declaration
30. (R) All automatics shall be initialized before being used

    This rule is checked using worst−case assumptions. This means that violations are reported not only for variables that are guaranteed to be uninitialized, but also for variables that are uninitialized on some execution paths.

31. (R) Braces shall be used in the initialization of arrays and structures
32. (R) Only the first, or all enumeration constants may be initialized
33. (R) The right hand operand of && or || shall not contain side effects
34. (R) The operands of a logical && or || shall be primary expressions
35. (R) Assignment operators shall not be used in Boolean expressions
36. (A) Logical operators should not be confused with bitwise operators
37. (R) Bitwise operations shall not be performed on signed integers
38. (R) A shift count shall be between 0 and the operand width minus 1

    This violation will only be checked when the shift count evaluates to a constant value at compile time.

39. (R) The unary minus shall not be applied to an unsigned expression
40. (A) ”sizeof” should not be used on expressions with side effects
41. x (A) The implementation of integer division should be documented
42. (R) The comma operator shall only be used in a ”for” condition
43. (R) Don’t use implicit conversions which may result in information loss

44. (A) Redundant explicit casts should not be used

45. (R) Type casting from any type to or from pointers shall not be used

46. (R) The value of an expression shall be evaluation order independent

    This rule is checked using worst−case assumptions. This means that a violation will be reported when a possible alias may cause the result of an expression to be evaluation order dependent.

47. (A) No dependence should be placed on operator precedence rules

48. (A) Mixed arithmetic should use explicit casting

49. (A) Tests of a (non−Boolean) value against 0 should be made explicit

50. (R) F.P. variables shall not be tested for exact equality or inequality

51. (A) Constant unsigned integer expressions should not wrap−around

52. (R) There shall be no unreachable code

53. (R) All non−null statements shall have a side−effect

54. (R) A null statement shall only occur on a line by itself

55. (A) Labels should not be used

56. (R) The ”goto” statement shall not be used

57. (R) The ”continue” statement shall not be used

58. (R) The ”break” statement shall not be used (except in a ”switch”)

59. (R) An ”if” or loop body shall always be enclosed in braces

60. (A) All ”if”, ”else if” constructs should contain a final ”else”

61. (R) Every non−empty ”case” clause shall be terminated with a ”break”

62. (R) All ”switch” statements should contain a final ”default” case

63. (A) A ”switch” expression should not represent a Boolean case

64. (R) Every ”switch” shall have at least one ”case”

65. (R) Floating−point variables shall not be used as loop counters

66. (A) A ”for” should only contain expressions concerning loop control

    A violation is reported when the loop initialization or loop update expression modifies an object that is not referenced in the loop test.

67. (A) Iterator variables should not be modified in a ”for” loop

68. (R) Functions shall always be declared at file scope

69. (R) Functions with variable number of arguments shall not be used

70. (R) Functions shall not call themselves, either directly or indirectly

    A violation will be reported for direct or indirect recursive function calls in the source file being checked. Recursion via functions in other source files, or recursion via function pointers is not detected.

71. (R) Function prototypes shall be visible at the definition and call

72. (R) The function prototype of the declaration shall match the definition

73. (R) Identifiers shall be given for all prototype parameters or for none

74. (R) Parameter identifiers shall be identical for declaration/definition

75. (R) Every function shall have an explicit return type

76. (R) Functions with no parameters shall have a ”void” parameter list

77. (R) An actual parameter type shall be compatible with the prototype

78. (R) The number of actual parameters shall match the prototype

79. (R) The values returned by ”void” functions shall not be used

80. (R) Void expressions shall not be passed as function parameters

81. (A) ”const” should be used for reference parameters not modified

82. (A) A function should have a single point of exit

83. (R) Every exit point shall have a ”return” of the declared return type

84. (R) For ”void” functions, ”return” shall not have an expression

85. (A) Function calls with no parameters should have empty parentheses

86. (A) If a function returns error information, it should be tested
A violation is reported when the return value of a function is ignored.

87. (R) #include shall only be preceded by other directives or comments

88. (R) Non-standard characters shall not occur in #include directives

89. (R) #include shall be followed by either <filename> or ”filename”

90. (R) Plain macros shall only be used for constants/qualifiers/specifiers

91. (R) Macros shall not be #define’d and #undef’d within a block

92. (A) #undef should not be used

93. (A) A function should be used in preference to a function-like macro

94. (R) A function-like macro shall not be used without all arguments

95. (R) Macro arguments shall not contain pre-preprocessing directives
A violation is reported when the first token of an actual macro argument is ’#’.

96. (R) Macro definitions/parameters should be enclosed in parentheses

97. (A) Don’t use undefined identifiers in pre-processing directives

98. (R) A macro definition shall contain at most one # or ## operator

99. (R) All uses of the #pragma directive shall be documented
This rule is really a documentation issue. The compiler will flag all #pragma directives as violations.

100. (R) ”defined” shall only be used in one of the two standard forms

101. (A) Pointer arithmetic should not be used

102. (A) No more than 2 levels of pointer indirection should be used
A violation is reported when a pointer with three or more levels of indirection is declared.

103. (R) No relational operators between pointers to different objects

In general, checking whether two pointers point to the same object is impossible. The compiler will only report a violation for a relational operation with incompatible pointer types.

104. (R) Non–constant pointers to functions shall not be used

105. (R) Functions assigned to the same pointer shall be of identical type

106. (R) Automatic address may not be assigned to a longer lived object

107. (R) The null pointer shall not be de–referenced

A violation is reported for every pointer dereference that is not guarded by a NULL pointer test.

108. (R) All struct/union members shall be fully specified

109. (R) Overlapping variable storage shall not be used

A violation is reported for every 'union' declaration.

110. (R) Unions shall not be used to access the sub–parts of larger types

A violation is reported for a 'union' containing a 'struct' member.

111. (R) Bit fields shall have type "unsigned int" or "signed int"

112. (R) Bit fields of type "signed int" shall be at least 2 bits long

113. (R) All struct/union members shall be named

114. (R) Reserved and standard library names shall not be redefined

115. (R) Standard library function names shall not be reused

x 116. (R) Production libraries shall comply with the MISRA C restrictions

x 117. (R) The validity of library function parameters shall be checked

118. (R) Dynamic heap memory allocation shall not be used

119. (R) The error indicator "errno" shall not be used

120. (R) The macro "offsetof" shall not be used

121. (R) <locale.h> and the "setlocale" function shall not be used

122. (R) The "setjmp" and "longjmp" functions shall not be used

123. (R) The signal handling facilities of <signal.h> shall not be used

124. (R) The <stdio.h> library shall not be used in production code

125. (R) The functions atof/atoi/atol shall not be used

126. (R) The functions abort/exit/getenv/system shall not be used

127. (R) The time handling functions of library <time.h> shall not be used

See also section 4.7, *C Code Checking: MISRA C*, in Chapter *Using the Compiler* of the User's manual.

Index