MB160-025-00-00
Doc. ver.: 1.1

PCP v2.5

C Compiler, Assembler, Linker Reference Manual

TASKING
Embedded software development from Altium

A publication of

Altium BV

Documentation Department

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

CONTENTS

TABLE OF CONTENTS

TASKING

CONTENTS

CONTENTS

MANUAL PURPOSE AND STRUCTURE

Windows Users

The documentation explains and describes how to use the PCP toolchain to program a PCP. The documentation is primarily aimed at Windows users. You can use the tools from the command line in a command prompt window.

Unix Users

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

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

Structure

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

First you need to install the software. This is described in Chapter 1, *Software Installation and Configuration*, 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 the Reference Manual to lookup specific options and details to make fully use of the PCP toolchain.

SHORT TABLE OF CONTENTS

Chapter 7: Object File Formats

Contains a description of the following object file formats:

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

Chapter 8: Linker Script Language

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

Chapter 9: 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 **cpcp** [**-?**] Both **cpcp** and **cpcp -?** are valid commands.
|	Separates items in a list. Read it as OR.
...	You can repeat the preceding item zero or more times.
,...	You can repeat the preceding item zero or more times, separating each item with a comma.

Example

cpcp [*option*]... *filename*

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

```
cpcp test.c
cpcp -g test.c
cpcp -g -E test.c
```

Not valid is:

```
cpcp -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.

Command line: type your input on the command line.

Reference: follow this reference to find related topics.

RELATED PUBLICATIONS

C Standards

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

MISRA-C

- MISRA-C:2004, Guidelines for the Use of the C Language in Critical Systems [MIRA Ltd, 2004]
  See also `http://www.misra-c.com`
- Guidelines for the Use of the C Language in Vehicle Based Software [MIRA Ltd, 1998]
  See also `http://www.misra.org.uk`

TASKING Tools

- PCP C Compiler, Assembler, Linker User's Manual
  [Altium, MA160-025-00-00]

CHAPTER 1

PCP C LANGUAGE

TASKING

1 CHAPTER

1.1 INTRODUCTION

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

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

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

1.2 DATA TYPES

The TASKING PCP C compiler **cpcp** supports the following data types:

Type	Keyword	Size (bit)	Align (bit)	Ranges
Boolean	`_Bool`	32	32	0 or 1
Character	`char` `signed char`	32	32	-2^{31} .. $2^{31}-1$
	`unsigned char`	32	32	0 .. $2^{32}-1$
Integral	`short` `signed short`	32	32	-2^{31} .. $2^{31}-1$
	`unsigned short`	32	32	0 .. $2^{32}-1$

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

	`unsigned int` `unsigned long`	32	32	0 .. $2^{32}-1$
	`enum`	32	32	-2^{31} .. $2^{31}-1$
	`long long` `signed long long`	32	32	-2^{31} .. $-2^{31}-1$
	`unsigned long long`	32	32	0 .. $2^{32}-1$
Pointer	pointer to data pointer to func	32	32	0 .. $2^{31}-1$ 0 .. $2^{32}-1$
Floating-Point	`float`	32	32	$-3.402e^{38}$ .. $-1.175e^{-38}$ $1.175e^{-38}$ .. 3.402^{e38}
	`double` `long double`	32	32	$-3.402e^{38}$ .. $-1.175e^{-38}$ $1.175e^{-38}$ .. 3.402^{e38}

Table 1-1: Fundamental Data Types

1.3 KEYWORDS

__asm()

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

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

instruction_template	Assembly instructions that may contain parameters from the input list or output list in the form: **%***parm_nr* [***.regnum***]
%*parm_nr*[***.regnum***]	Parameter number in the range 0 .. 7. With the optional ***.regnum*** you can access an individual register from a register pair or register quad.
output_param_list	[[**"=**[**&**]*constraint_char***"(***C_expression***)],**...]
input_param_list	[[**"***constraint_char***"(***C_expression***)],**...]
&	Says that an output operand is written to before the inputs are read, so this output must not be the same register as any input.
constraint _char	Constraint character: the type of register to be used for the *C_expression*.
C_expression	Any C expression. For output parameters it must be an *lvalue*, that is, something that is legal to have on the left side of an assignment.
register_save_list	[[**"***register_name***"],**...]
register_name	Name of the register you want to reserve.

Constraint character	Type	Operand	Remark
w	Word register	r0 .. r7	
number	Type of operand it is associated with	same as %*number*	Indicates that %*number* and *number* are the same register.

Table 1-2: Available input/output operand constraints

For more information on `__asm`, see section 2.5, *Using Assembly in the C Source*, in Chapter *PCP 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);
```

1.4 FUNCTION QUALIFIERS

inline
__noinline

With the `inline` keyword you tell the compiler to inline the function body instead of calling the function. Use the `__noinline` keyword to tell the compiler *not* to inline the function body. These keywords overrule smart inlining strategy that the compiler uses with compiler option **--optimize=+inline** (**-Oi**).

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

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

For more information see section 2.9.1, *Inlining Functions: inline*, in Chapter *PCP C Language* of the *User's Manual*.

__interrupt()

With the function qualifier `__interrupt` you can declare a function as interrupt function (interrupt service routine). This function qualifier takes one argument *CN*.The *CN* argument (Channel Number) is an 8 bit channel number in the range [0..255] that defines the channel entry table address and the context address. You should not use channel 0 (for channel number 0 no channel entry table address and no context address is defined.)

```
void __interrupt(CN) isr(void)
{
...
}
```

1.5 INTRINSIC FUNCTIONS

The TASKING PCP C compiler recognizes intrinsic functions.

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

You can use 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.

__dotdotdot__()

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

Variable argument "..." operator. Used in library function `va_start()`

__free()

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

Deallocates the memory pointed to by buffer. buffer must point to memory earlier alloacted by a call to __ `alloc()`. Same as library function `free()`

__nop()

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

Inserts a NOP instruction.

__get_return_address()

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

Used by the compiler in `retjmp()`.

__ld32_fpi()

```
unsigned long volatile __ld32_fpi ( unsigned long addr );
```

Load a 32-bit value from a 32-bit fpi address using the `ld.f` instruction with size=32. Returns a 32-bit value for fpi memory address.

Example:

```
unsigned int ld32( void )
{
return __ld32_fpi( (unsigned long)(&(P10_OUT.U)) );
}
```

generates:

```
ldl.iu  r5,@HI(0xf0003210)
ldl.il  r5,@LO(0xf0003210)
ld.f    r1,[r5], size=32
```

__st32_fpi()

```
void volatile __st32_fpi ( unsigned long addr,
                           unsigned long value );
```

Store a 32-bit value on a 32-bit fpi address using the **st.f** instruction with size=32.

Example:

```
#include <regtc1775b.sfr>
void st32( unsigned int value )
{
    __st32_fpi( (unsigned long)(&(P10_OUT.U)), value );
}
```

generates:

```
ldl.iu  r5,@HI(0xf0003210)
ldl.il  r5,@LO(0xf0003210)
st.f    r1,[r5], size=32
```

__exit()

```
void __exit( int srpn );
```

To allow rearbitration of interrupts, a 'voluntary exiting' scheme is supported via the intrinsic **__exit()**. This intrinsic generates an EXIT instruction with the following settings: EC=0, ST=0, INT=1, EP=1, cc_UC.

The R6.TOS is set for a PCP service request and the **sprn** value is loaded in R6.SPRN. It is your responsibility not to use this intrinsic in combinmation with the 'Channel Start at Base' mode.

The **srpn** value must be in range 0..255. If R6.SPRN is set to zero, it causes an illegal operation error on the PCP. When **srpn** is set to zero, no code is generated for loading R6.SPRN and the interrupt flag in the EXIT instruction is disabled (EXIT EC=0, ST=0, INT=0, EP=1, cc_UC).

1.6 PRAGMAS

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

For general information on pragmas see section 2.6, *Pragmas to Control the Compiler*, in Chapter *PCP C Language* of the *User's Manual*.

The syntax is:

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

or:

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

The compiler recognizes the following pragmas, other pragmas are ignored. Sometimes the resemblence of a pragma and a compiler option is so strong, that no explanation is given but instead is referred to the description of the corresponding compiler option.

#pragma alias *symbol=defined_symbol*

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

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

#pragma clear
#pragma noclear

Performs 'clearing' or no 'clearing' of non-initialized static/public variables.

#pragma compactmaxmatch *value*

Control the maximum size of a match.

See compiler option **--compact-max-size** in section *Compiler Options* in Chapter *Tool Options*.

#pragma extension isuffix

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

```
float 0.5i
```

***#pragma extern** symbol*

Normally, when you use the C keyword `extern`, the compiler generates an `.EXTERN` directive in the generated assembly source. However, if the compiler does not find any references to the extern symbol in the C module, it optimizes the assembly source by leaving the `.EXTERN` directive out.

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

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

#pragma inline
#pragma noinline
#pragma smartinline

See section 2.9.1, *Inlining Functions* of the *User's Manual*.

#pragma macro
#pragma nomacro

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

***#pragma maxcalldepth** value*

Control the maximum size of a match.

See compiler option **--max-call-depth** in section *Compiler Options* in Chapter *Tool Options*.

#pragma message "string"...

Print the message string(s) on standard output.

#pragma novector

Do not generate channel vectors and channel context. Same as compiler option --novector.

See compiler option **--novector** in section *Compiler Options* in Chapter *Tool Options*.

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

See section 4.3, *Compiler Optimizations* in Chapter *Using the Compiler* of the *User's Manual*. **`restore`** returns to the previous pragma level if the same pragma is used more than once.

#pragma protect
#pragma endprotect

Protect sections against linker optimizations. This excludes a section from unreferenced section removal and duplicate section removal by the linker.

#pragma section *section_type "section_name"*
#pragma endsection

See section 2.10, *Compiler Generated Sections* and compiler option **--rename-sections** in section *Compiler Options* in Chapter *Tool Options*.

#pragma source
#pragma nosource

See compiler option **-s** in section *Compiler Options* in Chapter *Tool Options*.

#pragma stdinc

Changes the behaviour of the #include directive. When set, compiler option **-I** and compiler option **--no-stdinc** are ignored.

#pragma linear_switch
#pragma jump_switch
#pragma binary_switch
#pragma smart_switch

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

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

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

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

`smart_switch` let the compiler decide the switch method used.

See section 2.11, *Switch Statement* of the *User's Manual*.

#pragma tradeoff *level*

Specify whether the used optizations should opimize for more speed (0), regardless of code size or for smaller code size (4), regardless of speed).

See also compiler option **-t (--tradeoff)** in section *Compiler Options* in Chapter *Tool Options*.

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

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

See also compiler option **-w (--no-warnings)** in section *Compiler Options* in Chapter *Tool Options*.

#pragma weak *symbol*

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

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

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

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

1.7 PREDEFINED MACROS

In addition to the predefined macros required by the ISO C standard, the TASKING PCP C compiler supports the predefined macros as defined in Table 1-3. The macros are useful to make conditional C code.

Macro	Description
__BIGENDIAN__	Expands to 0.
__BUILD__	Expands to the build number of the compiler: BRRRrrr. The B is the build number, RRR is the major branch number and rrr is the minor branch number. Examples: Build #134 -> 134000000 Build #22.1.4 -> 22001004
__CPCP__	Identifies the compiler. You can use this symbol to flag parts of the source which must be recognized by the **cpcp** assembler only. It expands to 1.
__CPU__	Expands to a string with the CPU supplied with the option **--cpu**. When no **--cpu** is supplied, this symbol is not defined.
__CORE__	Expands to a string with the core depending on the C compiler options option **--cpu**. The symbol expands to "pcp2" when the option **--cpu** is not specified.
__DATE__	Expands to the compilation date: "mmm dd yyyy".
__DOUBLE_FP__	Expands to 0. The PCP always treats a `double` as `float`.
__FILE__	Expands to the current source file name.
__LINE__	Expands to the line number of the line where this macro is called.
__REVISION__	Expands to the revision number of the compiler. Digits are represented as they are; characters (for prototypes, alphas, betas) are represented by -1. Examples: v1.0r1 -> 1, v1.0rb -> -1

Macro	Description
__SFRFILE__(*cpu*)	This macro expands to the filename of the used SFR file, including the < >. The *cpu* is the argument of the macro. For example, if **--cpu=tc1920b** is specified, the macro `__SFRFILE__(__CPU__)` expands to `__SFRFILE__(tc1920b)`, which expands to `<regtc1920b.sfr>`.
__SINGLE_FP__	Expands to 1. The PCP compiler always treats a `double` as `float`.
__STDC__	Identifies the level of ANSI standard. The macro expands to 1 if you set option **--language** (Control language extensions), otherwise expands to 0.
__STDC_HOSTED__	Always expands to 0, indicating the implementation is not a hosted implementation.
__STDC_VERSION__	Identifies the ISO C version number. Expands to 199901L for ISO C99 or 199409L for ISO C90.
__TASKING__	Identifies the compiler as a TASKING compiler. Expands to 1 if a TASKING compiler is used.
__TASKING_SFR__	Expands to 1 if TASKING `.sfr` files are used. Not defined when option **--no-tasking-sfr** is used.
__TIME__	Expands to the compilation time: ”hh:mm:ss”
__VERSION__	Expands to a number to identify the compiler version: Mmmm The M is the major version number and the mmm is a three-digit minor version number. Examples: 1.0 -> 1000 v12.3 -> 12003

Table 1-3: Predefined macros

C LANGUAGE

CHAPTER 2

LIBRARIES

TASKING

2 CHAPTER

2.1 INTRODUCTION

This chapter contains an overview of all library functions that you can call in your C source. This includes all functions of the standard C library (`libc.a`) and some functions of the floating-point library (`libfp.a` or `libfpt.a`).

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

The following libraries are included in the PCP (**cpcp**) toolchain. The control program **ccpcp** automatically selects the appropriate libraries depending on the specified options.

Library to link	Description
libc.a	C library (Some functions require the floating-point library. Also includes the startup code.)
libfp.a	Floating-point library (non-trapping)
libfpt.a	Floating-point library (trapping) (Control program option **--fp-trap**)

Table 2-1: Overview of libraries

2.2 LIBRARY FUNCTIONS

The tables in the sections below list all library functions, grouped per header file in which they are declared. Some functions are not completely implemented because their implementaion depends on the context where your application will run. These functions are for example all I/O related functions. Where possible, these functions are implemented using *file system simulation* (FSS). This system can be used by CrossView Pro to simulate an I/O environment which enables you to debug your application.

2.2.1 ASSERT.H

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

2.2.2 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]).

Ctype.h	Wctype.h	Description
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.
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.

2.2.3 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)

2.2.4 FCNTL.H

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

`open`	Opens a file a file for reading or writing. Calls `_open`. (*FSS implementation)*

2.2.5 FENV.H

Contains mechanisms to control the floating-point environment. The functions in this header file are not implemented.

`fegetenv`	Stores the current floating-point environment. (*Not implemented*)
`feholdexept`	Saves the current floating-point environment and installs an environment that ignores all floating-point exceptions. (*Not implemented*)
`fesetenv`	Restores a previously saved (fegetenv or feholdexcept) floating-point environment. (*Not implemented*)
`feupdateenv`	Saves the currently raised floating-point exceptions, restores a previousely saved floating-point environment and finally raises the saved exceptions. (*Not implemented*)
`feclearexcept`	Clears the current exception status flags corresponding to the flags specified in the argument. (*Not implemented*)
`fegetexceptflag`	Stores the current setting of the floating-point status flags. (*Not implemented*)
`feraiseexcept`	Raises the exceptions represented in the argument. As a result, other exceptions may be raised as well. (*Not implemented*)
`fesetexceptflag`	Sets the current floating-point status flags. (*Not implemented*)
`fetestexcept`	Returns the bitwise-OR of the exception macros corresponding to the exception flags which are currently set *and* are specified in the argument. (*Not implemented*)

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

FE_DIVBYZERO	FE_INEXACT	FE_INVALID
FE_OVERFLOW	FE_UNDERFLOW	FE_ALL_EXCEPT

fegetround	Returns the current rounding direction, represented as one of the values of the rounding direction macros. (*Not implemented*)
fesetround	Sets the current rounding directions. (*Not implemented*)

Currently no rounding mode macros are implemented.

2.2.6 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 2.2.12, *Math.h and Tgmath.h*.

2.2.7 FSS.H

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

Stdio.h	Description
_fss_break(*void*)	Buffer and breakpoint functions for CrossView Pro.
_fss_init(*fd*,*is_close*)	Opens file descriptors 0 (stdin), 1 (stdout) and 2 (stderr) and associates them with terminal window **FSS 0** of CrossView Pro.
_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 2.2.19, *Stdio.h*.

2.2.8 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 `strtoll`)
`uintmax_t strtoumax(const char * restrict nptr, char ** restrict endptr, int base);`	Convert string to maximum sized unsigned integer. (Compare `strtoull`)
`intmax_t wcstoimax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);`	Convert wide string to maximum sized integer. (Compare `wcstoll`)
`uintmax_t wcstoumax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);`	Convert wide string to maximem sized unsigned integer. (Compare `wcstoull`)

2.2.9 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  ^=
```

2.2.10 LIMITS.H

Contains the sizes of integral types, defined as macros.

2.2.11 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.

2.2.12 MATH.H AND TGMATH.H

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

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

Trigonometric functions

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

Math.h			Tgmath.h	Description
acosh	acoshf	acoshl	acosh	Returns the non-negative arc hyperbolic cosinus of x.
atanh	atanhf	atanhl	atanh	Returns the arc hyperbolic tangent of x.

Exponential and logarithmic functions

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

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

Rounding functions

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

Remainder after devision

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

frexp, ldexp, modf, scalbn, scalbln

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

Power and absolute-value functions

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

Manipulation functions: copysign, nan, nextafter, nexttoward

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

Positive difference, maximum, minimum

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

Error and gamma (Not implemented)

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

Comparison macros

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

Math.h	Tgmath.h	Description
`isgreater`	–	Returns the value of `(x) > (y)`
`isgreaterequal`	–	Returns the value of `(x) >= (y)`
`isless`	–	Returns the value of `(x) < (y)`
`islessequal`	–	Returns the value of `(x) <= (y)`
`islessgreater`	–	Returns the value of `(x) < (y) || (x) > (y)`
`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.

2.2.13 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()`.

2.2.14 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 behavior is used
`SIG_IGN`	The signal is ignored

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

2.2.15 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.

2.2.16 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
```

2.2.17 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`.

2.2.18 STDINT.H

See section 2.2.8, *inttypes.h and stdint.h*

2.2.19 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 (*FSS implementation)*
`fclose(name)`	Flushes the data stream and closes the specified file that was previously opened with fopen. (*FSS implementation)*

Stdio.h	Description
`fflush(`*`name`*`)`	If stream is an output stream, any buffered but unwritten date is written. Else, the effect is undefined. (*FSS implementation)*
`freopen(`*`name`*`,`*`mode`*`, `*`stream`*`)`	Similar to fopen, but rather then generating a new value of type FILE *, the existing value is associated with a new stream. (*FSS implementation)*
`setbuf(`*`stream`*`,`*`buffer`*`)`	If buffer is NULL, buffering is turned off for the stream. Otherwise, setbuf is equivalent to: `(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' or 'L'. 'h' indicates that the argument is to be treated as a short or unsigned short number. 'l' should be used if the argument is a long integer. 'L' indicates that the argument is a long double.

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

Character	Printed as
d, i	int, signed decimal
o	int, unsigned octal
x, X	int, unsigned hexadecimal in lowercase or uppercase respectively
u	int, unsigned decimal
c	int, single character (converted to unsigned char)

Character	Printed as
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 functions should be pointers to variables (in default memory) of the type which is specified in the format string.

The format string can contain :

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

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

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

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

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

Table 2-3: Scanf conversion characters

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

Stdio.h	Wchar.h	Description
scanf(*format*,...)	wscanf(*format*,...)	Performs a formatted read from stdin. Returns the number of items converted succesfully. (*FSS implementation)*
sscanf(*s, format*,...)	swscanf(*s, format*,...)	Performs a formatted read from the string *s*. Returns the number of items converted succesfully.
vfscanf(*stream, format,arg*)	vfwscanf(*stream, format,arg*)	Same as fscanf/fwscanf, but extra arguments are given as variable argument list *arg*. (See section 2.2.15, *stdarg.h*)
vscanf(*format, arg*)	vwscanf(*format, arg*)	Same as sscanf/swscanf, but extra arguments are given as variable argument list *arg*. (See section 2.2.15, *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 2.2.15, *stdarg.h*)
fputc(*c, stream*)	fputwc(*c, stream*)	Put character *c* onto the given *stream*. Returns EOF/WEOF on error. (*FSS implementation)*
putc(*c, stream*)	putwc(*c, stream*)	Same as fpuc/fputwc except that is implemented as a macro. (*FSS implementation)*
putchar(*c*, stdout)	putwchar(*c*, stdout)	Put character *c* onto the stdout stream. Returns EOF/WEOF on error. Implemented as macro. (*FSS implementation)*
fputs(*s, stream*)	fputws(*s, stream*)	Writes string *s* to the given *stream*. Returns EOF/WEOF on error. (*FSS implementation)*

Stdio.h	Wchar.h	Description
puts(*s)	-	Writes string *s* to the stdout stream. Returns EOF/WEOF on error. (*FSS implementation)*
fprintf(*stream*, *format*,...)	fwprintf(*stream*, *format*,...)	Performs a formatted write to the given *stream*. Returns EOF/WEOF on error. (*FSS implementation)*
printf(*format*, ...)	wprintf(*format*, ...)	Performs a formatted write to the stream stdout. Returns EOF/WEOF on error. (*FSS implementation)*
sprintf(*s, *format*,...)	-	Performs a formatted write to string *s*. Returns EOF/WEOF on error.
snprintf(*s, *n*, *format*,...)	swprintf(*s, *n*, *format*,...)	Same as sprintf, but n specifies the maximum number of characters (including the terminating null character) to be written.
vfprintf(*stream*, *format*,*arg*)	vfwprintf(*stream*, *format*,*arg*)	Same as fprintf/fwprintf, but extra arguments are given as variable argument list *arg*. (See section 2.2.15, *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 2.2.15, *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 2.2.15, *stdarg.h*)

Direct input/output

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

Random access

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

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

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

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

`rewind(`*`stream`*`)`	Sets the file position indicator for the *stream* to the beginning of the file. This function is equivalent to:
	`(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 2.2.3, *errno.h*)

2.2.20 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 2.2.11, *locale.h*).

Numeric conversions

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

Stdlib.h

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

Stdlib.h

```
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 2.2.14, *signal.h*).
`atexit(*`*`func`*`)`	*Func* points to a function that is called (without arguments) when the program normally terminates.
`exit(`*`status`*`)`	Causes normal program termination. Acts as if `main()` returns with status as the return value. Status can also be specified with the predefined macros EXIT_SUCCES or EXIT_FAILURE.
`_Exit(`*`status`*`)`	Same as `exit`, but not registered by the `atexit` function or signal handlers 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)`
`long labs(j)`
`long long llabs(j)`
	Compute the absolute value of an `int`, `long int`, and `long long int` *j* resepectively.

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

Multibyte/wide character and string conversions

`mblen(*s,n)`	Determines the number of bytes in the multi-byte character pointed to by *s*. At most *n* characters will be examined. (See also `mbrlen` in section 2.2.24, *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 2.2.24, *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 2.2.24, *wchar.h*)

2.2.21 STRING.H AND WCHAR.H

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

Copying and concatenation functions

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

Comparison functions

Stdio.h	Wchar.h	Description
`memcmp(*s1, *s2,n)`	`wmemcmp(*s1, *s2,n)`	Compares the first *n* characters of **s1* to the first *n* characters of **s2*. Returns < 0 if **s1* < **s2*, 0 if **s1* = = **s2*, or > 0 if **s1* > **s2*.
`strcmp(*s1,*s2)`	`wcscmp(*s1,*s2)`	Compares string **s1* to **s2*. Returns < 0 if **s1* < **s2*, 0 if **s1* = = **s2*, or > 0 if **s1* > **s2*.
`strncmp(*s1, *s2,n)`	`wcsncmp(*s1, *s2,n)`	Compares the first *n* characters of **s1* to the first *n* characters of **s2*. Returns < 0 if **s1* < **s2*, 0 if **s1* = = **s2*, or > 0 if **s1* > **s2*.
`strcoll(*s1,*s2)`	`wcscoll(*s1,*s2)`	Performs a local-specific comparison between string **s1* and string **s2* according to the LC_COLLATE category of the current locale. Returns < 0 if **s1* < **s2*, 0 if **s1* = = **s2*, or > 0 if **s1* > **s2*. (See section 2.2.11, *locale.h*)
`strxfrm(*s1, *s2,n)`	`wcsxfrm(*s1, *s2,n)`	Transforms (a local) string **s2* so that a comparison between transformed strings with `strcmp` gives the same result as a comparison between non-transformed strings with `strcoll`. Returns the transformed string **s1*.

Search functions

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

Miscellaneous functions

Stdio.h	Wchar.h	Description
`memset(*s,c,n)`	`wmemset(*s,c,n)`	Fills the first *n* bytes of **s* with character *c* and returns **s*.
`strerror(errno)`	-	Typically, the values for errno come from `int errno`. This function returns a pointer to the associated error message. (See also section 2.2.3, *errno.h*)
`strlen(*s)`	`wcslen(*s)`	Returns the length of string **s*.

2.2.22 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 2.2.11, *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
%% %

2.2.23 UNISTD.H

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

`access(*name,mode)`	Use the file system simulation of CrossView Pro to check the permissions of a file on the host. *mode* specifies the type of access and is a bit pattern constructed by a logical OR of the following values: `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 CrossView Pro 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 CrossView Pro to retrieve the current directory on the host. Returns the directory name. (*FSS implementation)*
`lseek(fd,offset, whence)`	Moves read-write file offset. Calls `_lseek()`. (*FSS implementation)*
`read(fd,*buff,cnt)`	Reads a sequence of characters from a file. This function calls `_read()`. (*FSS implementation)*
`stat(*name,*buff)`	Use the file system simulation feature of CrossView Pro to stat() a file on the host platform. (*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)*

2.2.24 WCHAR.H

Many functions in **wchar.h** represent the wide-character variant of other functions so these are discussed together. (See sections 2.2.19, *stdio.h*, 2.2.20, *stdlib.h*, 2.2.21, *strings.h* and 2.2.22, *time.h*).

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

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

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

`mbsinit(*ps)`	Determines whether the object pointed to by *ps*, is an initial conversion state. Returns a non-zero value if so.
`mbsrtowcs(*pwcs,**src, n,*ps)`	Restartable version of `mbstowcs`. See section 2.2.20, *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 2.2.20, *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 2.2.20, *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*.

2.2.25 WCTYPE.H

Most functions in **wctype.h** represent the wide-character variant of functions declared in **ctype.h** and are discussed in section 2.2.2, *ctype.h*. In addition, this header file provides extensible, locale specific functions and wide character classification.

`wctype(*property)`	Constructs a value of type `wctype_t` that describes a class of wide characters identified by the string **property*. If property identifies a valid class of wide characters according to the LC_TYPE category (see 2.2.11, *locale.h*) of the current locale, a non-zero value is returned that can be used as an argument in the `iswctype` function.
`iswctype(wc,desc)`	Tests whether the wide character *wc* is a member of the class represented by `wctype_t` *desc*. Returns a non-zero value if tested true.

Function	Equivalent to locale specific test
`iswalnum(wc)`	`iswctype(wc,wctype("alnum"))`
`iswalpha(wc)`	`iswctype(wc,wctype("alpha"))`
`iswcntrl(wc)`	`iswctype(wc,wctype("cntrl"))`
`iswdigit(wc)`	`iswctype(wc,wctype("digit"))`
`iswgraph(wc)`	`iswctype(wc,wctype("graph"))`

Function	Equivalent to locale specific test
`iswlower(`*wc*`)`	`iswctype(wc,wctype("lower"))`
`iswprint(`*wc*`)`	`iswctype(wc,wctype("print"))`
`iswpunct(`*wc*`)`	`iswctype(wc,wctype("punct"))`
`iswspace(`*wc*`)`	`iswctype(wc,wctype("space"))`
`iswupper(`*wc*`)`	`iswctype(wc,wctype("upper"))`
`iswxditig(`*wc*`)`	`iswctype(wc,wctype("xdigit"))`

`wctrans(*`*property*`)` Constructs a value of type `wctype_t` that describes a mapping between wide characters identified by the string **property*. If property identifies a valid mapping of wide characters according to the LC_TYPE category (see 2.2.11, *locale.h*) of the current locale, a non-zero value is returned that can be used as an argument in the `towctrans` function.

`towctrans(`*wc*`,`*desc*`)` Transforms wide character *wc* into another wide-character, described by *desc*.

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

CHAPTER 3

PCP ASSEMBLY LANGUAGE

TASKING

CHAPTER 3

3.1 INTRODUCTION

This chapter contains a detailed description of all built-in assembly functions directives and controls. For a description of the PCP instruction set, refer to the *PCP2 32-bit Single-Chip Microcontroller* [2000, Infineon].

3.2 BUILT-IN ASSEMBLY FUNCTIONS

3.2.1 OVERVIEW OF BUILT-IN ASSEMBLY FUNCTIONS

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

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

The following tables provide an overview of all built-in assembler functions. *expr* can be any assembly expression resulting in an integer value. Expressions are explained in section 3.6, *Assembly Expressions*, in chapter *Assembly Language* of the *User's Manual*.

Overview of mathematical functions

Function	Description
@ABS(*expr*)	Absolute value
@ACS(*expr*)	Arc cosine
@ASN(*expr*)	Arc sine
@AT2(*expr1*,*expr2*)	Arc tangent
@ATN(*expr*)	Arc tangent
@CEL(*expr*)	Ceiling function
@COH(*expr*)	Hyperbolic cosine
@COS(*expr*)	Cosine
@FLR(*expr*)	Floor function
@L10(*expr*)	Log base 10
@LOG(*expr*)	Natural logarithm
@MAX(*expr*,[,...,*exprN*])	Maximum value
@MIN(*expr*,[,...,*exprN*])	Minimum value
@POW(*expr1*,*expr2*)	Raise to a power
@RND()	Random value
@SGN(*expr*)	Returns the sign of an expression as –1, 0 or 1
@SIN(*expr*)	Sine
@SNH(*expr*)	Hyperbolic sine
@SQT(*expr*)	Square root
@TAN(*expr*)	Tangent
@TNH(*expr*)	Hyperbolic tangent
@XPN(*expr*)	Exponential function (raise e to a power)

Overview of conversion functions

Function	Description
@CVF(*expr*)	Convert integer to floating-point
@CVI(*expr*)	Convert floating-point to integer
@FLD(*base*,*value*, *width*[,*start*])	Shift and mask operation
@FRACT(*expr*)	Convert floating-point to 32-bit fractional
@SFRACT(*expr*)	Convert floating-point to 16-bit fractional
@LNG(*expr*)	Concatenate to double word
@LUN(*expr*)	Convert long fractional to floating-point
@RVB(*expr1*[,*expr2*])	Reverse order of bits in field
@UNF(*expr*)	Convert fractional to floating-point

Overview of string functions

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

Overview of macro functions

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

Overview of address calculation functions

Function	Description
@DPTR(*expr*)	PCP only: returns bits 6–13 of the pcpdata address
@HI(*expr*)	Returns upper 16 bits of expression value
@INIT_R7(*start*,*dptr*,*flags*)	PCP only: returns the 32–bit value to initialize R7
@LO(*expr*)	Returns lower 16 bits of expression value
@LSB(*expr*)	Get least significant byte of a word
@MSB(*expr*)	Get most significant byte of a word

Overview of assembler mode functions

Function	Description
@ASPCP()	Returns the name of the PCP assembler executable
@CPU(*string*)	Test if CPU type is selected
@DEF('*symbol*'|*symbol*)	Returns 1 if symbol has been defined
@EXP(*expr*)	Expression check
@INT(*expr*)	Integer check
@LST()	LIST control flag value

3.2.2 DETAILED DESCRIPTION OF BUILT-IN ASSEMBLY FUNCTIONS

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

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

Example:

```
AVAL  .SET @ABS(-2.1)   ; AVAL = 2
```

@ACS(*expression****)***

Returns the arc cosine of *expression* as a floating-point value in the range zero to pi. The result of *expression* must be between -1 and 1.

Example:

```
ACOS   .SET   @ACS(-1.0)   ;ACOS = 3.1415926535897931
```

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

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

Example:

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

@ASN(*expression****)***

Returns the arc sine of *expression* as a floating-point value in the range -pi/2 to pi/2. The result of *expression* must be between -1 and 1.

Example:

```
ARCSINE   .SET   @ASN(-1.0)    ;ARCSINE = -1.570796
```

@ASPCP()

Returns the name of the PCP assembler executable. This is 'aspcp' for the PCP assembler.

Example:

```
ANAME:   .byte   @ASPCP()   ; ANAME = 'aspcp'
```

@AT2(*expr1,expr2****)***

Returns the arc tangent of *expr1/expr2* as a floating-point value in the range -pi to pi. *Expr1* and *expr2* must be separated by a comma.

Example:

```
ATAN2   .EQU   @AT2(-1.0,1.0)   ;ATAN2 = -0.7853982
```

@ATN(*expression****)***

Returns the arc tangent of *expression* as a floating-point value in the range -pi/2 to pi/2.

Example:

```
ATAN  .SET  @ATN(1.0)   ;ATAN = 0.78539816339744828
```

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

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

Example:

```
.DEFINE  ID  "@CAT('PCP','-assembler')"   ;ID =
'PCP-assembler'
```

@CEL(*expression****)***

Returns a floating-point value which represents the smallest integer greater than or equal to *expression*.

Example:

```
CEIL  .SET  @CEL(-1.05)     ;CEIL = -1.0
```

@CNT()

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

Example:

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

@COH(*expression****)***

Returns the hyperbolic cosine of *expression* as a floating-point value.

Example:

```
HYCOS  .EQU  @COH(VAL)      ;compute hyperbolic cosine
```

@COS(*expression****)***

Returns the cosine of *expression* as a floating-point value.

Example:

```
.WORD  -@COS(@CVF(COUNT)*FREQ) ;compute cosine value
```

@CPU(*string****)***

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

Example:

```
IF @CPU("pcp")                ;PCP specific part
```

@CVF(*expression****)***

Converts the result of *expression* to a floating-point value.

Example:

```
FLOAT  .SET  @CVF(5)          ;FLOAT = 5.0
```

@CVI(*expression****)***

Converts the result of *expression* to an integer value. This function should be used with caution since the conversions can be inexact (e.g., floating-point values are truncated).

Example:

```
INT  .SET  @CVI(-1.05)        ;INT = -1
```

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

Returns an integer 1 if *symbol* has been defined, 0 otherwise. *symbol* can be any symbol or label not associated with a `.MACRO` or `.SDECL` directive. If *symbol* is quoted it is looked up as a `.DEFINE` symbol; if it is not quoted it is looked up as an ordinary symbol or label.

Example:

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

@DPTR(*expression****)***

Returns bits 6-13 of the pcpdata address provided. This is equivalent to `(((expression >> 6) & 0xff) << 8)`.

Example:

```
ldl.il r7,@DPTR(pcp_data_a0)
```

@EXP(*expression****)***

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

Example:

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

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

@FLD(*base,value,width[,start]****)***

Shift and mask *value* into *base* for *width* bits beginning at bit *start*. If *start* is omitted, zero (least significant bit) is assumed. All arguments must be positive integers and none may be greater than the target word size. Returns the shifted and masked value.

Example:

```
VAR1 .EQU @FLD(0,1,1)   ;turn bit 0 on, VAR1=1
VAR2 .EQU @FLD(0,3,1)   ;turn bit 0 on, VAR2=1
VAR3 .EQU @FLD(0,3,2)   ;turn bits 0 and 1 on, VAR3=3
VAR4 .EQU @FLD(0,3,2,1) ;turn bits 1 and 2 on, VAR4=6
VAR5 .EQU @FLD(0,1,1,7) ;turn eighth bit on, VAR5=0x80
```

@FLR(*expression****)***

Returns a floating-point value which represents the largest integer less than or equal to *expression*.

Example:

```
FLOOR   .SET  @FLR(2.5)          ;FLOOR = 2.0
```

@FRACT(*expression****)***

This function returns the 32-bit fractional representation of the floating-point expression. The expression must be in the range [-1,+1>.

Example:

```
.WORD  @FRACT(0.1), @FRACT(1.0)
```

@HI(*expression****)***

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

Example:

```
mov.u   d2,#@LO(COUNT)
addih   d2,d2,#@HI(COUNT)   ; upper 16 bits of COUNT
```

@INIT_R7(*start,dptr,flags****)***

Returns the 32-bit value needed to initialize R7. This is equivalent to `(`*`start`*`<<16) + (((dptr&0x3fff)>>6)<<8) + (`*`flags`*` & 0xff)`.

Example:

```
.word @init_r7(start_0,pcp_data_0,7)
```

@INT(*expression****)***

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

Example:

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

@L10(*expression****)***

Returns the base 10 logarithm of *expression* as a floating-point value. *expression* must be greater than zero.

Example:

```
LOG    .EQU  @L10(100.0)        ;LOG = 2
```

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

Returns the length of *string* as an integer.

Example:

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

@LNG(*expr1,expr2****)***

Concatenates the 16-bit *expr1* and *expr2* into a 32-bit word value such that *expr1* is the high half and *expr2* is the low half.

Example:

```
LWORD   .WORD  @LNG(HI,LO)          ;build long word
```

@LO(*expression****)***

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

Example:

```
mov.u   d2,#@LO(COUNT)          ;lower 16 bits of COUNT
addih   d2,d2,#@HI(COUNT)
```

@LOG(*expression****)***

Returns the natural logarithm of *expression* as a floating-point value. *expression* must be greater than zero.

Example:

```
LOG     .EQU  @LOG(100.0)       ;LOG = 4.605170
```

@LSB(*expression****)***

Returns the least significant byte of the result of the *expression*. *expression* is interpreted as a half word (16 bit).

Example:

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

@LST()

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

Example:

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

@LUN(*expression****)***

Converts the 32-bit *expression* to a floating-point value. *expression* should represent a binary fraction.

Example:

```
DBLFRC1 .EQU  @LUN(0x40000000)  ;DBLFRC1 = 0.5
DBLFRC2 .EQU  @LUN(3928472)     ;DBLFRC2 = 0.007354736
DBLFRC3 .EQU  @LUN(0xE0000000)  ;DBLFRC3 = -0.75
```

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

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

Example:

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

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

Returns the greatest of *expr1*,...,*exprN* as a floating-point value.

Example:

```
MAX:    .BYTE @MAX(1,-2.137,3.5)  ;MAX = 3.5
```

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

Returns the least of *expr1*,...,*exprN* as a floating-point value.

Example:

```
MIN:    .BYTE @MIN(1,-2.137,3.5)  ;MIN = -2.137
```

@MSB(*expression****)***

Returns the most significant byte of the result of the *expression*. *expression* is interpreted as a half word (16 bit).

Example:

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

@MXP()

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

Example:

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

@POS(*str1,str2[,start]****)***

Returns the position of *str2* in *str1* as an integer, starting at position *start*. If *start* is not given the search begins at the beginning of *str1*. If the *start* argument is specified it must be a positive integer and cannot exceed the length of the source string. Note that the first position in a string is position 0.

Example:

```
ID    .EQU  @POS('PCP-assembler','assembler')    ;ID =
4
ID2   .EQU  @POS('ABCDABCD','B',2)    ;ID2 = 5
```

@POW(*expr1,expr2****)***

Returns *expr1* raised to the power *expr2* as a floating-point value. *expr1* and *expr2* must be separated by a comma.

Example:

```
BUF   .EQU  @CVI(@POW(2.0,3.0))      ;BUF = 8
```

@RND()

Returns a random value in the range 0.0 to 1.0.

Example:

```
SEED  .EQU   @RND()           ;save initial SEED value
```

@RVB(*expr1,expr2****)***

Reverse the order of bits in *expr1* delimited by the number of bits in *expr2*. If *expr2* is omitted the field is bounded by the target word size. Both expressions must be 16-bit integer values.

Example:

```
VAR1 .SET @RVB(0x200)   ;reverse all bits, VAR1=0x40
VAR2 .SET @RVB(0xB02)   ;reverse all bits, VAR2=0x40D0
VAR3 .SET @RVB(0xB02,2) ;reverse bits 0 and 1,
                        ;VAR3=0xB01
```

@SCP(*str1,str2****)***

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

Example:

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

@SFRACT(*expression****)***

This function returns the 16-bit fractional representation of the floating-point expression. The expression must be in the range [-1,+1>.

Example:

```
.WORD  @SFRACT(0.1), @SFRACT(1.0)
```

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

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

Example:

```
VAR1  .SET @SGN(-1.2e-92)   ;VAR1 = -1
VAR2  .SET @SGN(0)          ;VAR2 =  0
VAR3  .SET @SGN(28.382)     ;VAR3 =  1
```

@SIN(*expression****)***

Returns the sine of *expression* as a floating-point value.

Example:

```
.WORD   @SIN(@CVF(COUNT)*FREQ)   ;compute sine value
```

@SNH(*expression****)***

Returns the hyperbolic sine of *expression* as a floating-point value.

Example:

```
HSINE .EQU  @SNH(VAL)          ;hyperbolic sine
```

@SQT(*expression****)***

Returns the square root of *expression* as a floating-point value. *expression* must be positive.

Example:

```
SQRT1  .EQU  @SQT(3.5)         ;SQRT1 = 1.870829
SQRT2  .EQU  @SQT(16)          ;SQRT2 = 4
```

@SUB(*string,expression1,expression2****)***

Returns the substring from *string* as a string. *Expression1* is the starting position within *string*, and *expression2* is the length of the desired string. The assembler issues an error if either *expression1* or *expression2* exceeds the length of *string*. Note that the first position in a string is position 0.

Example:

```
.DEFINE  ID  "@SUB('PCP-assembler',4,9)"  ;ID =
'assembler'
```

@TAN(*expression****)***

Returns the tangent of *expression* as a floating-point value.

Example:

```
TANGENT .SET @TAN(1.0)         ;TANGENT = 1.5574077
```

@TNH(*expression****)***

Returns the hyperbolic tangent of *expression* as a floating-point value.

Example:

```
HTAN  .SET  @TNH(1)            ;HTAN = 0.76159415595
```

@UNF(*expression****)***

Converts *expression* to a floating-point value. *expression* should represent a 16-bit binary fraction.

Example:

```
FRC    .EQU  @UNF(0x4000)        ;FRC = 0.5
```

@XPN(*expression****)***

Returns the exponential function (base e raised to the power of *expression*) as a floating-point value.

Example:

```
EXP    .EQU  @XPN(1.0)           ;EXP = 2.718282
```

3.3 ASSEMBLER DIRECTIVES AND CONTROLS

3.3.1 OVERVIEW OF ASSEMBLER DIRECTIVES

Assembler directives are grouped in the following categories:

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

The following tables provide an overview of all assembler directives.

Overview of assembly control directives

Directive	Description
.COMMENT	Start comment lines
.DEFINE	Define substitution string
.END	End of source program
.FAIL	Programmer generated error message
.INCLUDE	Include file
.MESSAGE	Programmer generated message
.ORG	Initialize memory space and location counters to create a nameless section
.SDECL	Declare a section with name, type and attributes
.SECT	Activate a declared section
.UNDEF	Undefine DEFINE symbol
.WARNING	Programmer generated warning

Overview of symbol definition directives

Directive	Description
.ALIAS	Create an alias for a symbol
.EQU	Assign permanent value to a symbol
.EXTERN	External symbol declaration
.GLOBAL	Global section symbol declaration
.LOCAL	Local symbol declaration
.NAME	Specify name of original C source file
.SET	Set temporary value to a symbol
.SIZE	Set size of symbol in the ELF symbol table
.TYPE	Set symbol type in the ELF symbol table
.WEAK	Mark symbol as 'weak'

Overview of data definition / storage allocation directives

Directive	Description
.ACCUM	Define 64–bit constant of 18 + 46 bits format
.ALIGN	Define alignment
.ASCII / .ASCIIZ	Define ASCII string without / with ending NULL byte
.BYTE	Define constant byte
.FLOAT / .DOUBLE	Define a 32–bit / 64–bit floating–point constant
.FRACT / .SFRACT	Define a 16–bit / 32–bit constant fraction
.SPACE	Define storage
.WORD / .HALF	Define a word / half–word constant

Overview of macro and conditional assembly directives

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

Overview of debug directives

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

3.3.2 DETAILED DESCRIPTION OF ASSEMBLER DIRECTIVES

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

.ACCUM

Syntax

[*label*:] **.ACCUM** *expression*[,*expression*]...

Description

With the `.ACCUM` directive (Define 64-bit Constant) the assembler allocates and initializes two words of memory (64 bits) for each argument. Use commas to separate multiple arguments.

An *expression* can be:

- a fractional fixed point expression (range $[-2^{17}, 2^{17}>$)
- NULL (indicated by two adjacent commas: ,,)

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

If the evaluated expression is out of the range $[-2^{17}, 2^{17}>$, the assembler issues a warning and saturates the fractional value.

Example

```
ACC:  .ACCUM  0.1,0.2,0.3
```

Related information

.SPACE (Define storage)

.FRACT / **.SFRACT** (Define 32-bit / 16-bit constant fraction)

.ALIAS

Syntax

alias-name **.ALIAS** *function-name*

Description

With the `.ALIAS` directive you can create an alias of a symbol. The C compiler generates this directive when you use the #pragma alias.

Example

```
_malloc .ALIAS ___hmalloc
```

Related information

-

.ALIGN

Syntax

.ALIGN *expression*

Description

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

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

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

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

A label is not allowed before this directive.

Example

```
.ALIGN 16          ; the assembler aligns
add r5,r1,cc_uc    ; this instruction at 16 bytes and
                   ; fills the 'gap' with NOP instructions

.ALIGN 12          ; WRONG: not a power of two, the
add r5,r1,cc_uc    ; assembler aligns this instruction at
                   ; 16 bytes and issues a warning
```

Related information

-

.ASCII/.ASCIIZ

Syntax

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

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

Description

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

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

Example

```
STRING:  .ASCII  "Hello world"

STRINGZ: .ASCIIZ "Hello world"
```

With the `.BYTE` directive you can obain exactly the same effect:

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

Related information

.SPACE (Define storage)

.BYTE (Define a constant byte)
.WORD / **.HALF** (Define a word / halfword)

.BYTE

Syntax

[*label*] **.BYTE** *argument*[**,***argument*]...

Description

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

An *argument* can be:

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

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

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

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

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

```
.BYTE 'R'          ; = 0x52
.BYTE 'AB',,'D'    ; = 0x41420043
```

Example

```
TABLE   .BYTE   'two',0,'strings',0
CHARS   .BYTE   'A','B','C','D'
```

Related information

.SPACE (Define storage)

.ASCII / **.ASCIIZ** (Define ASCII string without/with ending NULL)
.WORD / **.HALF** (Define a word / halfword)

.CALLS

Syntax

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

Description

Create a flow graph reference between *caller* and *callee*. With this information the linker can build a call graph and calculate stack size. *Caller* and *Callee* are names of functions.

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

A label is not allowed before this directive.

Example

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

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

Related information

-

.COMMENT

Syntax

.COMMENT *delimiter*
.
.
delimiter

Description

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

A label is not allowed before this directive.

Example

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

Related information


-

.DEFINE

Syntax

.DEFINE *symbol string*

Description

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

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

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

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

A label is not allowed before this directive.

Example

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

```
.DEFINE  LEN  '32'
```

then the source line below:

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

would be transformed by the assembler to the following:

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

Related information

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

.DUP / .ENDM

Syntax

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

Description

The sequence of source lines between the `.DUP` and `.ENDM` directives will be duplicated by the number specified by the integer *expression*. If the expression evaluates to a number less than or equal to 0, the sequence of lines will not be included in the assembler output. The expression result must be an absolute integer and cannot contain any forward references to address labels (labels that have not already been defined). You can nest the `.DUP` directive to any level.

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

Example

Consider the following source input statements,

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

This is expanded as follows:

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

Related information

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

.DUPA / .ENDM

Syntax

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

Description

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

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

Example

Consider the following source input statements,

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

This is expanded as follows:

```
.BYTE  12
.BYTE  VALUE  ; results in a warning
.BYTE  32
.BYTE  34
```

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

Related information

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

.DUPC / .ENDM

Syntax

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

Description

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

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

Example

Consider the following source input statements,

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

This is expanded as follows:

```
.BYTE  1
.BYTE  2
.BYTE  3
```

Related information

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

.DUPF / .ENDM

Syntax

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

Description

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

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

Example

Consider the following source input statements,

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

This is expanded as follows:

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

Related information

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

.END

Syntax

.END [*expression*]

Description

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

The *expression* is only permitted here for compatibility reasons. It is ignored during assembly.

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

A label is not allowed before this directive.

Example

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

Related information

-

.EQU

Syntax

symbol **.EQU** *expression*

Description

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

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

Example

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

```
A_D_PORT  .EQU  0x4000
```

You cannot redefine the symbol `A_D_PORT` after this.

Related information

.SET (Set temporary value to a symbol)

.EXITM

Syntax

.EXITM

Description

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

A label is not allowed before this directive.

Example

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

Related information

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

.EXTERN

Syntax

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

Description

With the `.EXTERN` directive (External Symbol Declaration) you specify that the list of symbols is referenced in the current module, but is not defined within the current module. These symbols must either have been defined outside of any module or declared as globally accessible within another module with the `.GLOBAL` directive.

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

A label is not allowed before this directive.

Example

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

Related information

.GLOBAL (Global symbol declaration)
.LOCAL (Local symbol declaration)

.FAIL

Syntax

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

Description

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

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

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

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

A label is not allowed before this directive.

Example

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

This results in the error:

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

Related information

.MESSAGE (Programmer Generated Message),
.WARNING (Programmer Generated Warning)

.FLOAT/.DOUBLE

Syntax

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

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

Description

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

An *expression* can be:

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

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

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

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

Examples

```
FLT:   .FLOAT  12.457,+0.27E-13

DBL:   .DOUBLE  12.457,+0.27E-13
```

Related information

.SPACE (Define storage)

.FRACT/.SFRACT

Syntax

[*label*:] **.FRACT** *expression*[,*expression*]...

[*label*:] **.SFRACT** *expression*[,*expression*]...

Description

With the `.FRACT` or `.SFRACT` directive the assembler allocates and initializes one word of memory (32 bits) or a halfword (16 bits) for each argument. Use commas to separate multiple arguments.

An *expression* can be:

- a fractional fixed point expression (range [-1, +1>)
- NULL (indicated by two adjacent commas: ,,)

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

If the evaluated argument is out of the range [-1, +1> , the assembler issues a warning and saturates the fractional value.

Example

```
FRCT:   .FRACT   0.1,0.2,0.3

SFRCT:  .SFRACT  0.1,0.2,0.3
```

Related information

.SPACE (Define storage)

.ACCUM (Define 64-bit constant fraction in 18+46 bits format)

.GLOBAL

Syntax

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

Description

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

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

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

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

A label is not allowed before this directive.

Example

```
        .SDECL  ".pcpdata.data",DATA
        .SECT   ".pcpdata.data"
        .GLOBAL LOOPA   ; LOOPA will be globally
                        ; accessible by other modules
LOOPA .HALF   0x100     ; assigns the value 0x100 to LOOPA
```

Related information

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

.IF / .ELIF / .ELSE / .ENDIF

Syntax

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

Description

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

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

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

A label is not allowed before this directive.

Example

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

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

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

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

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

```
aspcp -DDEMO -DTEST=0 test.src
```

Related information

-

.INCLUDE

Syntax

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

Description

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

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

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

1. The current directory if you used the **'***filename***'** construction.

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

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

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

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

A label is not allowed before this directive.

Example

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

Related information

Assembler option **–I** (Add directory to include file search path) in section 5.2, *Assembler Options*, of Chapter *Tool Options*.

.LOCAL

Syntax

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

Description

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

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

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

A label is not allowed before this directive.

Example

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

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

Related information

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

.MACRO / .ENDM

Syntax

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

Description

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

The definition of a macro consists of three parts:

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

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

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:

```
CONSTD  .MACRO  reg,value                    ;header
    ldl.iu reg,@hi(value)                    ;body
    ldl.il reg,@lo(value)
    .ENDM                                    ;terminator
```

The macro call:

```
    .SDECL '.pcptext', code
    .SECT '.pcptext'
    CONSTD  r5,0x12345678
    .END
```

The macro expands as follows:

```
    ldl.iu  r5,@hi(0x12345678)
    ldl.il  r5,@lo(0x12345678)
```

Related information

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

Section 3.10, *Macro Operations*, in Chapter *Assembly Language* of the *User's Manual*.

.MESSAGE

Syntax

.MESSAGE [{*string* | *exp*}[,{*string* | *exp*}]...]

Description

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

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

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

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

A label is not allowed before this directive.

Example

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

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

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

Related information

.FAIL (Programmer Generated Error)
.WARNING (Programmer Generated Warning)

.NAME

Syntax

.NAME *string*

Description

The `.SOURCE` directive specifies the name of the original C source module.

This directive is generated by the C compiler. You do not need to specify this directive in hand-written assembly.

Example

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

Related information

-

.ORG

Syntax

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

Description

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

This directive uses the following arguments:

abs-loc Initial value to assign to the run-time location counter. *abs-loc* must be an absolute expression. If *abs_loc* is not specified, then the value is zero.

sect_type An optional section type:

code	code section
data	data section

attribute An optional section attribute:

Code attibutes:

init	section is copied from ROM to RAM at startup
noread	section can be executed from but not read

Data attibutes:

noclear	section is not cleared during startup
max	data overlay with other parts with the same name, is implicit a type of 'noclear'
rom	data section remains in ROM

A label is not allowed before this directive.

Example

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

; define a relocatable nameless section
.org

; define a relocatable data section
.org   ,data
```

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

Related information

.SDECL (Declare section name and attributes)
.SECT (Activate a declared section)

.PMACRO

Syntax

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

Description

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

A label is not allowed before this directive.

Example

```
.PMACRO   MAC1,MAC2
```

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

Related information

.MACRO (Define Macro)

.SDECL

Syntax

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

Description

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

This directive uses the following arguments:

type: A section type:

code	code section
data	data section
debug	debug section

attribute: An optional section attribute:

Code attibutes:

init section is copied from ROM to RAM at startup

noread section can be executed from but not read

cluster ("*group*"**)**
Cluster code sections with companion debug sections. Used by the linker during removal of unreferenced sections. The name must be unique for this module (not for the application).

protect Tells the linker to exclude a section from unreferenced section removal and duplicate section removal.For each grou

Data attibutes:

noclear section is not cleared during startup

max data overlay with other parts with the same name, is implicit a type of 'noclear'

rom data section remains in ROM

group ("*group***")**
Used to group sections, for example for placing in the same page.

cluster ("*name***")**
Cluster data sections with companion debug sections. Used by the linker during removal of unreferenced sections. The name must be unique for this module (not for the application).

protect Tells the linker to exclude a section from unreferenced section removal and duplicate section removal.

overlay ("*name***")**
The linker can overlay sections that have the same pool *name*.

Debug attributes:

cluster ("*name***")**
Cluster code sections with companion debug sections. Used by the linker during removal of unreferenced sections. The name must be unique for this module (not for the application).

Sections with attribute `noclear` are not zeroed at startup. This is a default attribute for `data` sections. You can only use this attribute with a `data` type section. This attribute is only useful with BSS sections, which are cleared at startup by default.

The attribute `init` defines that the `code` section contains initialization data, which is copied from ROM to RAM at program startup.

Sections with the attribute `rom` contain data to be placed in ROM. This ROM area is not executable.

When `data` sections with the same name occur in different object modules with the attribute `max`, the linker generates a section with a size that is the largest of the sizes in the individual object modules. The attribute `max` only applies to `data` sections.

The *name* of a section can have a special meaning for locating sections. The name of code sections should always start with "`.pcptext`", the name of data sections should always start with "`.pcpdata`".

Note that the compiler uses the following name convention:

prefix.module-name.function-or-object-name

Examples:

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

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

.SECT (Activate a declared section)
.ORG (Initialize a nameless section)

.SECT

Syntax

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

Description:

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

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

Examples:

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

.SDECL (Declare a section with name, type and attributes)
.ORG (Initialize a nameless 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 another `.SET`. directive.

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

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

Example

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

Related information

..EQU (Assign permanent value to a symbol)

.SIZE

Syntax

.SIZE *symbol*, *expression*

Description

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

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

Example

```
main:  .type   func
       .      ; function main
       .
       ret16
main_function_end:
       .size   main,main_function_end-main
```

Related information

.TYPE (Set Symbol Type)

.SPACE

Syntax

[*label*] **.SPACE** *expression*

Description

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

With *expression* you specify the number of MAUs (Minimum Addressable Units) 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). For the PCP assembler **aspcp**, the MAU size is 2 bytes for pcp code sections and 4 bytes for pcp data sections.

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

Example

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

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

Related information

.ASCII / **.ASCIIZ** (Define ASCII string without/with ending NULL)
.BYTE (Define a constant byte)
.FLOAT / **.DOUBLE** (Define a 32-bit / 64-bit floating-point constant)
.WORD / **.HALF** (Define a word / halfword)

.TYPE

Syntax

symbol **.TYPE** *typeid*

Description

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

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

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

Example

```
Afunc    .TYPE    FUNC
```

Related information

.SIZE (Set Symbol Size)

.UNDEF

Syntax

.UNDEF *symbol*

Description

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

A label is not allowed before this directive.

Example

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

Related information

.DEFINE (Define Substitution String)

.WARNING

Syntax

.WARNING [{*string* | *exp*}[**,**{*string* | *exp*}]...]

Description

With the **.WARNING** directive (Programmer Generated Warning) you tell the assembler to output a warning message during the assembling process.

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

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

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

A label is not allowed before this directive.

Example

```
.WARNING  'parameter too large'
```

This results in the warning:

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

Related information

.FAIL (Programmer Generated Error),
.MESSAGE (Programmer Generated Message)

.WEAK

Syntax

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

Description

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

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

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

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

Example

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

Related information


-

.WORD/.HALF

Syntax

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

[*label*] **.HALF** *argument*[**,***argument*]...

Description

With the `.WORD` or `.HALF` directive the assembler allocates and initializes one word (32 bits) or a halfword (16 bits) of memory for each *argument*.

An *argument* can be:

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

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

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

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

```
.WORD 'R'           ; = 0x52
.WORD 'ABCD'        ; = 0x41424344

.HALF 'R'           ; = 0x52
.HALF 'AB'          ; = 0x4142
.HALF 'ABCD'        ; = 0x4142
                        0x4344
```

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

Examples

```
WRD:  .WORD  14,1635,0x34266243,'ABCD'

HLF:  .HALF  14,1635,0x2662,'AB'
```

With the `.BYTE` directive you can obain exactly the same effect:

```
WRD:   .BYTE   14,0,0,0,1635%256,6,0,0,
               0x43,0x62,0x26,0x34,'D','C','B','A'

HLF:   .BYTE   14,0,1635%256,6,0x62,0x26,'B','A'
```

Related information

.SPACE (Define storage)

.ASCII / **.ASCIIZ** (Define ASCII string without/with ending NULL)
.BYTE (Define a constant byte)

3.3.3 OVERVIEW OF ASSEMBLER CONTROLS

The following tables provide an overview of all assembler controls.

Overview of assembler listing controls

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

Overview of miscellaneous assembler controls

Function	Description
$DEBUG ON / OFF	Generation of symbolic debug ON/OFF
$DEBUG "*flags*"	Select debug information
$HW_ONLY	Prevent substitution of assembly instructions by smaller or faster instructions
$IDENT LOCAL / GLOBAL	Assembler treats labels by default as local or global
$OBJECT	Alternative name for the generated object file
$WARNING OFF [*num*]	Suppress all or some warnings

3.3.4 DETAILED DESCRIPTION OF ASSEMBLER CONTROLS

The assembler recognizes both upper and lower case for controls.

$DEBUG ON / OFF

Syntax

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

Description

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

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

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

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

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

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

Example

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

Related option

Assembler option **-g** (Select debug information) in section 5.2, *Assembler Options*, of Chapter *Tool Options*.

Related information

-

$HW_ONLY

Syntax

$HW_ONLY

Description

Normally the assembler replaces instructions by other, smaller or faster instructions. For example, the instruction `jeq d0,#0,label1` is replaced by `jz d0,label1`.

With the `$HW_ONLY` control you instruct the assembler to encode all instruction as they are. The assembler does not substitute instructions with other, faster or smaller instructions.

Example

```
;begin of source
$HW_ONLY   ; the assembler does not substitute
           ; instructions with other, smaller or
           ; faster instructions.
```

Related option

Assembler option **-Og** (Allow generic instructions) in section 5.2, *Assembler Options*, of Chapter *Tool Options*.

Related information

-

$IDENT

Syntax

$IDENT LOCAL
$IDENT GLOBAL

Description

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

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

Example

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

Related option

Assembler option **–i** (Treat labels by default local / global) in section 5.2, *Assembler Options*, of Chapter *Tool Options*.

Related information

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

$LIST ON / OFF

Syntax

$LIST ON

.
. ; assembly source lines
.

$LIST OFF

Description

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

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

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

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

Example

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

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

The assembler generates a list file with the following lines:

```
      .SDECL ".pcptext.code",code
      .SECT  ".pcptext.code"
      ...  ; source line in list file
   $LIST ON
      ...  ; source line also in list file
      .END
```

Related option

Assembler option **-l** (Generate list file) in section 5.2, *Assembler Options*, of Chapter *Tool Options*.

Related information

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

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

$LIST *flags*

Syntax

Begin of assembly file

$LIST "*flags***"**

Description

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

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

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

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

Example

To exclude assembly files with controls from the list file:

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

Related option

Assembler option **–L** (List file formatting options) in section 5.2, *Assembler Options*, of Chapter *Tool Options*.

Related information

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

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

$OBJECT

Syntax

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

Description

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

Example

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

Related option

Assembler option **-o** (Define output filename) in section 5.2, *Assembler Options*, of Chapter *Tool Options*.

Related information

-

$PAGE

Syntax

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

Description

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

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

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

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

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

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

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

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

Example

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

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

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

Related option

 -

Related information

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

Assembler option **–l** (Generate list file) in Section 5.2, *Assembler Options*, of Chapter *Tool Options*.

Assembler option **–L** (List file formatting options) in Section 5.2, *Assembler Options*, of Chapter *Tool Options*.

$PRCTL

Syntax

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

Description

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

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

You can specify the following arguments:

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

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

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

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

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

Example

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

Related option

-

Related information

Assembler option **–l** (Generate list file) in Section 5.2, *Assembler Options*, of Chapter *Tool Options*.

$STITLE

Syntax

$STITLE "*title*"

Description

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

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

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

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

Example

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

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

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

Related option

-

Related information

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

Assembler option **–l** (Generate list file) in Section 5.2, *Assembler Options*, of Chapter *Tool Options*.

$TITLE

Syntax

$TITLE "*title*"

Description

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

By default, the title is empty.

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

Example

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

The header of the list file will now be:

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

Related option


-

Related information


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

$WARNING OFF

Syntax

$WARNING OFF
$WARNING OFF *number*

Description

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

- By default, all warnings are reported.
- If you specify this option but without numbers, all warnings are suppressed.
- If you specify this option with a number, only the specified warning is suppressed.

Example

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

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

Related option

Assembler option **-w** (Suppress some or all warnings) in section 5.2, *Assembler Options*, of Chapter *Tool Options*.

Related information

-

CHAPTER 4

RUN-TIME ENVIRONMENT

TASKING

4 CHAPTER

4.1 INTRODUCTION

This chapter describes the startup code used by the TASKING PCP C compiler, the stack layout and the heap; and the floating-point arithmetic.

4.2 STARTUP CODE

Just as the PCP is part of the TriCore processor, a PCP application is part of a TriCore application. However, the PCP application runs as an interrupt service routine which is activated by the TriCore application.

The PCP C startup code acts as a 'wrapper' which places the PCP `main()` application into an interrupt service routine on interrupt channel 1.

When this interrupt is activated, it executes in parallel with the TriCore application and returns the exit code of the PCP `main()` application after finishing execution..

See also section 6.6, *Linking the C Startup Code* in Chapter *Using the Linker* of the *User's Manual*.

4.3 STACK USAGE

The stack is used for parameter passing, allocation of automatics, temporary storage and storing the function return address. The compiler uses a static stack. Overlay sections are generated by the compiler to contain the stack objects. The overlay sections are overlayed by the linker using a call graph.

4.4 HEAP ALLOCATION

The heap is only needed when you use one or more of the dynamic memory management library functions: `malloc()`, `calloc()`, `free()` and `realloc()`. The heap is a reserved area in memory. Only if you use one of the memory allocation functions listed above, the linker automatically allocates a heap, as specified in the linker script file with the keyword `pcp_heap`.

A special section called `pcp_heap` is used for the allocation of the heap area. The size of the heap is defined in the linker script file (`tc_arch.lsl` in directory `include.lsl`) with the macro `PCPHEAP`, which results in a section called `pcp_heap`. The linker defined labels `(_PCP_)_lc_ub_heap` and `(_PCP_)_lc_ue_heap` (begin and end of heap) are used by the library function `sbrk()`, which is called by `malloc()` when memory is needed from the heap.

The special `pcp_heap` section is only allocated when its linker labels are used in the program.

4.5 FLOATING-POINT ARITHMETIC

Floating-point arithmetic support for the compiler **cpcp** is included in the software as a separate set of libraries. During linking you have to specify the desired floating-point library (trapping or non-trapping) after the C library.

The **cpcp** compiler only supports single-precision floating-point arithmetic. The single-precision floating-point complies to the ISO/IEC 9899 standard.

To ensure portability of floating-point arithmetic, floating-point arithmetic for the compiler **cpcp** has been implemented complying with the IEEE-754 standard for floating-point arithmetic. See the *IEEE Standard Binary for Floating-Point Arithmetic* document [IEEE Computer Society, 1985] for more details on the floating-point arithmetic definitions. This document is referred to as IEEE-754 in this manual.

It is possible to intercept floating-point exceptional cases and, if desired, handle them with an application defined exception handler. The intercepting of floating-point exceptions is referred to as 'trapping'. Examples of how to install a trap handler are included.

4.5.1 COMPLIANCE WITH IEEE-754

The level to which the floating-point implementation complies with the IEEE-754 standard, depends on the choosen configuration.

All floating-point calculations are executed using the 'round to nearest (even)' rounding mode, since this is required by ANSI-C 89. This is conform IEEE-754. Because there are no double precision floating-point hardware instructions, an emulating library is always needed for double precision calculation.

Compliance with IEEE-754: Trapping emulation library

The following implementation issues for the trapping floating-point library are important:

- subnormals are not supported. This is conform the PCP hardware design.
- when converting floats to integers, the value is truncated. This complies with ANSI-C 89 and ISO-C 99, but does not comply with IEEE-754, since the current rounding mode is 'round to nearest (even)'.
- when a converted float overflows the target integer type, a predictable value is assigned to the target integer.

Compliance with IEEE-754: Non-trapping emulation library

The following implementation issues for the non-trapping floating-point library are important:

- when calculating with floats, rounding is done to the nearest integer (rounding towards infinity when equally near).
- there is no distinction between -0 and +0
- when an operand of a calculation is a NaN, Inf or subnormal, the result is undefined.
- when the result of a calculation would be a subnormal, the result is 0.
- whenever a NaN or Inf would be the result of a calculation, the result is undefined
- when converting single precision floats to integers, rounding is done to the nearest integer (rounding towards infinity when equally near).
- when converting double precision floats to integers, the value is truncated. This is similar to the trapping emulation library.
- when a converted float overflows the target integer type, the value is saturated to MAX_INT or MIN_INT.

4.5.2 SPECIAL FLOATING-POINT VALUES

Below is a list of special, IEEE-754 defined, floating-point values as they can occur during run-time.

Special value		Sign	Exponent	Mantissa
+0.0	(Positive Zero)	0	all zeros	all zeros
-0.0	(Negative Zero)	1	all zeros	all zeros
+INF	(Positive Infinite)	0	all ones	all zeros
-INF	(Negative Infinite)	1	all ones	all zeros
NaN	(Not a number)	0	all ones	not all zeros

Table 4-1: Special floating-point values

4.5.3 TRAPPING FLOATING-POINT EXCEPTIONS

Four floating-point libraries are delivered:

Library to link	Description
libfp.a	Floating-point library (non-trapping). This is the default.
libfpt.a	Floating-point library (trapping) (Control program option **--fp-trap**)

Table 4-2: Floating-point libraries

The control program **ccpcp** automatically selects the appropriate libraries depending on the specified PCP. By specifying the **--fp-trap** option to the control program **ccpcp**, the trapping type floating-point library is linked into your application. If this option is not specified, the floating-point library without trapping mechanism is used.

IEEE-754 Trap Handler

In the IEEE-754 standard a trap handler is defined, which is invoked on (specified) exceptional events, passing along much information about the event. To install your own trap handler, use the library call `_fp_install_trap_handler`. When installing your own exception handler, you must select on which types of exceptions you want to have your handler invoked, using the function call `_fp_set_exception_mask`. See below for more details on the floating-point library exception handling function interface.

SIGFPE Signal Handler

In ANSI-C the regular approach of dealing with floating-point exceptions is by installing a so-called signal handler by means of the ANSI-C library call `signal`. If such a handler is installed, floating-point exceptions cause this handler to be invoked. To have the signal handler for the **SIGFPE** signal actually become operational with the provided floating-point libraries, a (very) basic version of the IEEE-754 exception handler must be installed (see example below) which will raise the desired signal by means of the ANSI-C library function call `raise`. For this to be achieved, the function call `_fp_install_trap_handler` is present. When installing your own exception handler, you will have to select on which types of exceptions you want to receive a signal, using the function call `_fp_set_exception_mask`. See further below for more details on the floating-point library exception handling function interface.

There is no way to specify any information about the context or nature of the exception to the signal handler. Just that a floating-point exception occurred can be detected. See therefor the IEEE-754 trap handler discussion above if you want more control over floating-point results.

Example:

```
#include <float.h>
#include <signal.h>

static void pass_fp_exception_to_signal(
                  _fp_exception_info_t *info )
{
      info;     /* suppress parameter not used warning */

      /* cause SIGFPE signal to be raised */

      raise( SIGFPE );
      /*
       * now continue the program
       * with the unaltered result
       */
}
```

4.5.4 FLOATING-POINT TRAP HANDLING API

For purposes of dealing with floating-point arithmetic exceptions, the following library calls are available:

```
#include <float.h>

int   _fp_get_exception_mask( void );
void  _fp_set_exception_mask( int );
```

A pair of functions to get or set the mask which controls which type of floating-point arithmetic exceptions are either ignored or passed on to the trap handler. The types of possible exception flag bits are defined as:

```
EFINVOP
EFDIVZ
EFOVFL
EFUNFL
EFINEXCT
```

while,

```
EFALL
```

is the OR of all possible flags. See below for an explanation of each flag.

```
#include <float.h>

int   _fp_get_exception_status( void );
void  _fp_set_exception_status( int );
```

A pair of functions for examining or presetting the status word containing the accumulation of all floating-point exception types which occurred so far. See the possible exception type flags above.

```
#include <float.h>

void  _fp_install_trap_handler( void (*)
                    (_fp_exception_info_t * ) );
```

This function call expects a pointer to a function, which in turn expects a pointer to a structure of type `_fp_exception_info_t`. The members of `_fp_exception_info_t` are:

exception

This member contains one of the following (numerical) values:

EFINVOP
EFDIVZ
EFOVFL
EFUNFL
EFINEXCT

operation

This member contains one of the following numbers:

_OP_ADDITION
_OP_SUBTRACTION
_OP_COMPARISON
_OP_EQUALITY
_OP_LESS_THAN
_OP_LARGER_THAN
_OP_MULTIPLICATION
_OP_DIVISION
_OP_CONVERSION

source_format
destination_format

Numerical values of these two members are:

_TYPE_SIGNED_CHARACTER
_TYPE_UNSIGNED_CHARACTER
_TYPE_SIGNED_SHORT_INTEGER
_TYPE_UNSIGNED_SHORT_INTEGER
_TYPE_SIGNED_INTEGER
_TYPE_UNSIGNED_INTEGER
_TYPE_SIGNED_LONG_INTEGER
_TYPE_UNSIGNED_LONG_INTEGER
_TYPE_FLOAT
_TYPE_DOUBLE

```
operand1    /* left side of binary or */
            /* right side of unary */
operand2    /* right side for binary */
result
```

These three are of the following type, to receive and return a value of arbitrary type:

```
typedef union _fp_value_union_t
{
      char  c;
      unsigned char  uc;
      short  s;
      unsigned short  us;
      int  i;
      unsigned int  ui;
      long  l;
      unsigned long  ul;
      float  f;
#if ! _SINGLE_FP
      double  d;
#endif
}
_fp_value_union_t;
```

The following table lists all the exception code flags, the corresponding error description and result:

Error Description	Exception Flag	Default Result with Trapping
Invalid Operation	EFINVOP	NaN
Division by zero	EFDIVZ	+INF or -INF
Overflow	EFOVFL	+INF or -INF
Underflow	EFUNFL	zero
Inexact	EFINEXT	undefined
INF Infinite which is the largest absolute floating-point number, which is always: -INF < every finite number < +INF NAN Not a Number, a symbolic entity encoded in floating-point format.		

Table 4-3: Exception Type Flag Codes

To ensure all exception types are specified, you can specify **EFALL** to a function, which is the binary OR of all above enlisted flags.

CHAPTER 5

TOOL OPTIONS

TASKING

CHAPTER 5

5.1 COMPILER OPTIONS

This section lists all compiler options.

Short and long option names

Options have both short and long names. Short option names always begin with a single minus (**-**) character, long option names always begin with two minus (**--**) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

Options can have flags or suboptions. To switch a flag 'on', use a lowercase letter or a *+longflag*. To switch a flag off, use an uppercase letter or a *-longflag*. Separate *longflags* with commas. The following two invocations are equivalent:

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

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

-? (--help)

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
typedefs	Show the list of predefined typedefs

Description

Displays an overview of all command line options. With an argument you can specify which extended information is shown.

Example

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

```
cpcp -?
cpcp --help
cpcp
```

The following invocation displays a list of the available pragmas:

```
cpcp --help=pragmas
```

Related information

-

–A (––language)

Command line syntax

–A[*flags*]

––language=[*flags*]

You can set the following flags:

g/**G**	(**+**/**–gcc**)	Enable a number of gcc extensions
p/**P**	(**+**/**–comments**)	Allow C++ style (`//`)comments in ISO C90
x/**X**	(**+**/**–strings**)	Relaxed const check for string literals

Default

–AGpx

Description

With this option you control the language extensions the compiler can accept. By default the PCP C compiler allows all language extensions, except for gcc extensions.

The option **–A** (**––language**) without flags is the equivalent of **–AGPX** and *disables* all language extensions.

With **–Ag** you tell the compiler to enable the following gcc languages extensions:

- The identifier __FUNCTION__ expands to the current function name.
- Alternative syntax for variadic macros.
- Alternative syntax for designated initializers.
- Allow zero sized arrays.
- Allow empty struct/union.
- Allow empty initializer list.
- Allow initialization of static objects by compound literals.
- The middle operand of a `? :` operator may be omitted.
- Allow a compound statement inside braces as expression.
- Allow arithmetic on void pointers and function pointers.
- Allow a range of values after a single `case` label.
- Additional preprocessor directive `#warning`.
- Allow comma operator, conditional operator and cast as lvalue.

- An inline function without "`static`" or "`extern`" will be global.
- An "`extern inline`" function will not be compiled on its own.

For an exact description of these gcc extensions, please refer to the gcc info pages (**info gcc**).

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

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

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

Example

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

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

Related information

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

--align-stack

Command line syntax

--align-stack=*value*

Default

--align-stack=64

Description

Align static stack sections with a size smaller than or equal to *value* so that these sections are not located over a page boundary. This optimization saves code becuase the DPTR does not have to be reload when it already contains the right page number.

The disadvantage is that data space is spilled for the alignment. The alignment must be a power of two in the range [1..64]. 1 equals to no alignment optimizations. The default value 64 turns on alignment optimization for all static sections.

Example

To specify another alignment value:

```
cpcp --align-stack=32 test.c

cpcp --align-stack=20 test.c \\ not allowed, value is
                                not a power of 2.
```

Related information

-

-C (--cpu)

Command line syntax

-C*cpu*

--cpu=*cpu*

Description

With this option you define the target processor for which you create your application. Make sure you choose a target processor *with* pcp!

The compiler always includes the register file `reg`*`cpu`*`.sfr`.

Example

To compile the file `test.c` for the `TC1920B` processor (a processor with pcp) and use the SFR file `regtc1920b.sfr`:

```
cpcp -Ctc1920b test.c
cpcp --cpu=tc1920b test.c
```

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

Related information

Assembler option **-C** (Select CPU)
Control program option **-C** (Use SFR definitions for CPU)

Section 4.4, *Calling the Compiler*, in Chapter *Using the Compiler* of the *User's Manual*.

–c (––iso)

Command line syntax

–c{90|99}

––iso={90|99}

Default

–c99

Description

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

Example

To select the ISO C90 standard on the command line:

```
cpcp -c90 test.c
cpcp --iso=90 test.c
```

Related information

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

--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 reports any warnings and/or errors.

Example

To check for syntax errors, without generating code:

```
cpcp --check test.c
```

Related information

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

--compact-max-size

Command line syntax

--compact-max-size=*value* (Default: 200)

Description

This option is related to the compiler optimization **-Or** (Code compaction or reverse inlining). Code compaction is the opposite of inlining functions: large sequences of code that occur more than once, are transformed into a function. This reduces code size (possibly at the cost of execution speed).

However, in the process of finding sequences of matching instructions, compile time and compiler memory usage increase quadratically with the number of instructions considered for code compaction. With this option you tell the compiler to limit the number of matching instructions it considers for code compaction.

Example

To limit the maximum number of instructions in functions that the compiler generates during code compaction:

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

Related information

Compiler option **-Or** (Common subexpression elimination)

--core

Command line syntax

--core=*core*

You can specify the following *core* arguments:

pcp1
pcp1_5
pcp2

Description

With this option you specify the core architecture for a custom target processor for which you create your application. By default the PCP toolset derives the core from the processor you selected.

Example

To check for syntax errors, without generating code:

```
cpcp --core pcp1 test.c
```

Related information

Compiler option **-C** (Use SFR definitions for CPU)

–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 limit of the operating system, you can define the macros in an *option file* which you then must specify to the compiler with the option **–f** *file*.

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

Example

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

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

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

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

cpcp --define=DEMO test.c
cpcp --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.

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

Related information

Compiler option **–U** (Remove preprocessor macro)
Compiler option **–f** (Specify an option file)

--diag

Command line syntax

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

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

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

Description

With this option the compiler displays a description and explanation of the specified error message(s) on `stdout` (usually the screen). The compiler does not compile any files.

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:

```
cpcp --diag=282
```

This results in the following message and explanation:

```
E282: unterminated comment

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

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

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

Related information

Section 4.8, *C Compiler Error Messages*, in Chapter *Using the Compiler* of the *User's Manual*.

–E (––preprocess)

Command line syntax

–E[*flags*]

––preprocess[**=***flags*]

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

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

Description

With this option you tell the compiler to preprocess the C source. On the command line, the compiler sends the preprocessed file to stdout. To capture the information in a file, specify an output file with the option **–o**.

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

With **–Em** the compiler generates dependency lines that can be used in a Makefile. The preprocessor output is discarded. The default target name is the basename of the input file, with the default object file suffix.

With **–Ep** you tell the preprocessor to strip the #line source position information (lines starting with `#line`). These lines are normally processed by the assembler and not needed in the preprocessed output. When you leave these lines out, the output is easier to read.

Example

```
cpcp –EcMP test.c –o test.pre

cpcp ––preprocess=+comments,–make,–noline test.c
    ––output=test.pre
```

The compiler preprocesses the file `test.c` and sends the output to the file `test.pre`. Comments are included but no dependencies are generated and the line source position information is not stripped from the output file.

Related information

–

--error-file

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:

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

Related information

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

-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 files you want to specify. With this option you specify the option file to the compiler.

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

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

Format of an option file

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

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

'This has a double quote " embedded'

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

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

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

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

Example

Suppose the file `myoptions` contains the following lines:

```
-Ctc1920b
-s
test.c
```

Specify the option file to the compiler:

```
cpcp -f myoptions
cpcp --option-file=myoptions
```

This is equivalent to the following command line:

```
cpcp -Ctc1920b -s test.c
```

Related information


-

-g (--debug-info)

Command line syntax

-g[*suboption*]

--debug-info[**=***suboption*]

You can set the following suboptions:

a	(**all**)	Emit full symbolic debug information
c	(**call-frame**)	Emit DWARF call-frame information only
d	(**default**)	Emit default symbolic debug information

Default: **--debug-info** (same as **--debug-info=default**)

Description

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

Call-frame information only

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.

Full 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.

Example

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

```
cpcp -g test.c
cpcp --debug test.c
```

Related information

-

-H (--include-file)

Command line syntax

-H*file*,...

--include-file=*file*,...

Description

With this option you include one extra file at the beginning of each C source file, before other includes. This is the same as specifying `#include "file"` at the beginning of each of your C sources.

Example

```
cpcp -Hstdio.h test1.c test2.c
cpcp --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*.

–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.

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 CPCPINC when the product was installed.
4. The default directory `$(PRODDIR)\include` (unless you specified option **`--no-stdinc`**).

Example

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

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

You can call the compiler as follows:

```
cpcp -Imyinclude test.c
cpcp --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 option **--no-stdinc** (Skip standard include files directory)

Section 4.5, *How the Compiler Searches Include Files*, in Chapter *Using the Compiler* of the *User's Manual*.

Section 1.3.1, *Configuring the Command Line Environment*, in Chapter *Software Installation* of the *User's Manual*.

--inline

Command line syntax

--inline

Description

With this option you instruct the compiler to inline *all* functions without the function qualifier `__noinline`, regardless whether they have the keyword `inline` or not. This option has the same effect as a `#pragma inline` at the beginning of the source file.

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

Example

To inline all functions:

```
cpcp --inline test.c
```

Related information

Compiler option **–Or** (Optimization: code compaction)

--inline-max-incr / --inline-max-size

Command line syntax

--inline-max-incr=*percentage*

--inline-max-size=*threshold*

Default

--inline-max-incr=25

--inline-max-size=10

Description

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

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

With the option **--inline-max-size** you can specify the maximum size of functions that the compiler inlines as part of the optimization process. The compiler always inlines *all* functions that are smaller than the specified *threshold*. The threshold is measured in compiler internal units and the compiler uses this measure to decide which functions are small enough to inline. The default threshold is 10.

After the compiler has inlined all functions that have the function qualifier `inline` and all functions that are smaller than the specified threshold, the compiler looks whether it can inline more functions without increasing the code size too much. With the option **--inline-max-incr** you can specify how much the code size is allowed to increase. By default, this is 35% which means that the compiler continues inlining functions until the resulting code size is 35% larger than the original size.

Example

```
cpcp --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 2.9.1, *Inlining Functions*, in Chapter *PCP C Language* of the *User's Manual*.

-k (--keep-output-files)

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 **mkpcp**. If the erroneous files are not removed, the make utility may process corrupt files on a subsequent invocation.

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

Example

```
cpcp -k test.c
```

When an error occurs during compilation, the generated output file `test.src` will *not* be removed.

Related information

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

--misrac

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

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

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

Related information

See Chapter 9 *MISRA-C Rules* for a list of all supported MISRA-C rules.

Compiler option **--misrac-advisory-warnings**
Compiler option **--misrac-required-warnings**
Compiler option **--misrac-version**

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

--misrac-advisory-warnings / --misrac-required-warnings

Command line syntax

--misrac-advisory-warnings

--misrac-required-warnings

Description

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

Example

```
cpcp --misrac=all --misrac-advisory-warnings test.c
```

The compiler generates an error for each MISRA-C rule violation in file `test.c`. If one of the advisory rules is violated, a warning instead of an error is generated.

Related information

See Chapter 9 *MISRA-C Rules* for a list of all supported MISRA-C rules.

Compiler option **--misrac**

--misrac-version

Command line syntax

--misrac-version={1998|2004}

Description

MISRA-C rules exist in two versions: MISRA-C:1998 and MISRA-C:2004. By default, the C source is checked against the MISRA-C:2004 rules. With this option you can specify to check against the MISRA-C:1998 rules.

Related information

See Chapter 9 *MISRA-C Rules* for a list of all supported MISRA-C rules.

Compiler option **--misrac**

-n (--stdout)

Command line syntax

-n

--stdout

Description

With this option you tell the compiler to send the output to stdout (usually your screen). No files are created.

This option is for example useful to quickly inspect the output or to redirect the output to other tools.

Example

```
cpcp -n test.c
```

The compiler sends the output (normally `test.src`) to stdout and does not create the file `test.src`.

Related information

-

--no-clear

Command line syntax

--no-clear

Description

Normally global/static variables are cleared at program startup. With this option you tell the compiler to generate code to prevent non-initialized global/static variables from being cleared at program startup.

This option applies to constant as well as non-constant variables.

Related information

Pragmas **clear**/**noclear**

--no-stdinc

Command line syntax

--no-stdinc

Description

With this option you tell the compiler not to look in the default `include` directory relative to the installation directory, when searching for include files. This way the compiler only searches in the include file search paths you specified.

Example

```
cpcp -Imyinclude --no-stdinc test.c
cpcp --include-directory=myinclude --no-stdinc test.c
```

Same example as with **-I** option, but now the compiler does not search in the default `include` directory.

Related information

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

--no-tasking-sfr

Command line syntax

--no-tasking-sfr

Description

Normally, the compiler includes a special function register (SFR) file before compiling. The compiler automatically selects the SFR file belonging to the target you select on the **Processor definition** page of the Processor options (compiler option **-C**).

With this option the compiler does *not* include the register file `reg`*`cpu`*`.sfr` as based on the selected target processor.

Use this option if you want to use your own set of SFR files.

Example

```
cpcp -Ctc1920b --no-tasking-sfr test.c
```

The register file `regtc1920b.sfr` is not included.

Related information

Compiler option **-C** (Use SFR definitions for CPU)

--novector

Command line syntax

--novector

Description

With this option you tell the compiler not to generate code for channel vectors and channel context.

Example

```
cpcp --novector test.c
```

No code for channel vectors and channel context is generated.

Related information


-

–O (––optimize)

Command line syntax

–O[*flags*]
––optimize[**=***flags*]

You can set the following flags:

a/**A**	(**+**/**–coalesce**)	Coalescer: remove unnecessary moves
b/**B**	(**+**/**–ipro**)	Interprocedural register optimizations
c/**C**	(**+**/**–cse**)	Common subexpression elimination
e/**E**	(**+**/**–expression**)	Expression simplification
f/**F**	(**+**/**–flow**)	Control flow optimization and code reordering
g/**G**	(**+**/**–glo**)	Generic assembly optimizations
h/**H**	(**+**/**–partition**)	Automatic memory partition
i/**I**	(**+**/**–inline**)	Function inlining
l/**L**	(**+**/**–loop**)	Loop transformations
o/**O**	(**+**/**–forward**)	Forward store
p/**P**	(**+**/**–propagate**)	Constant propagation
r/**R**	(**+**/**–compact**)	Code compaction (reverse inlining)
s/**S**	(**+**/**–subscript**)	Subscript strength reduction
y/**Y**	(**+**/**–peephole**)	Peephole optimizations

Use the following options for predefined sets of flags:

–O0 (**––optimize=0**) — No optimization.
Alias for: **–OABCEFGHILOPRSY**

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 same order as written in the source code, associative and commutative properties are not used.

–O1 (**––optimize=1**) — Debug purpose optimization
Alias for: **–OabcefghILOPRSy**

Enables optimizations that do not affect the debug-ability of the source code. Use this level when you are developing/debugging new source code.

–O2 (**––optimize=2**) — Release purpose optimization (default)
Alias for: **–OabcefghIlopRSy**

Enables more optimizations to reduce code size 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 have already been debugged. This is the default optimization level.

-O3 (**--optimize=3**) Aggressive optimization
Alias for: **-OabcefghiloprSy**

Enables aggressive global optimization techniques. Although in theory debugging is still possible, the relation between source code and generated instructions is complex and hard to understand. Use this level to compress your program into the system memory, or to decrease execution time to meet your real-time requirements.

Default

-O2

Description

With this option you can control the level of optimization. If you do not use this option, the default optimization level is *medium optimization* (option **-O2** or **-O** or **-OabcefghIlopRSy**).

When you use this option to specify a set of optimizations, you can overrule these settings in your C source file with `#pragma optimize` *`flag`* and `#pragma endoptimize`.

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

Example

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

```
cpcp test.c

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

cpcp -O test.c
cpcp --optimize test.c
```

```
cpcp -OacefgIklMopsvwy test.c
cpcp --optimize=+coalesce,+cse,+expression,+flow,
     +glo,-inline,+schedule,+loop,-simd,+forward,
     +propagate,+subscript,+ifconvert,+pipeline,
     +peephole test.c
```

Related information

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

```
#pragma optimize flag
#pragma endoptimize
```

Section 4.3, *Compiler Optimizations*, in Chapter *Using the Compiler* of the *User's Manual*.

-o (--output)

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

```
cpcp -o output.src test.c
cpcp --output=output.src test.c
```

The compiler creates the file `output.src` for the compiled file `test.c`.

Without the option **-o**, the compiler uses the names of the input file and creates `test.src`.

Related information

-

–R (––rename-sections)

Command line syntax

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

–R[*type***=**]*format_string*[,[*type***=**]*format_string*]...

Description

In case a module must be loaded at a fixed address, or a data section needs a special place in memory, you can use this option to generate different section names. You can then use this unique section name in the linker script file for locating.

With the memory type you select which sections are renamed. The matching sections will get the specified format_string for the section name. The format string can contain characters and may contain the following format specifiers:

{attrib}	section attributes,separated by underscores
{module}	module name
{name}	object name, name of variable or function
{type}	section type

Instead of this option you can also uses the pragmas section/endsection in the C source.

Example

To rename sections of memory type *data* to pcp_test_*variable_name*:

```
cpcp --rename-sections=data=pcp_{module}_{name}
     test.c
```

Related information

Section 2.10, *Compiler Generated Sections*, in Chapter *PCP C Language* of the *User's Manual*.

–s (––source)

Command line syntax

–s

––source

Description

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

Example

```
cpcp –s test.c
```

The output file `test.src` contains the original C source lines as comments, besides the generated assembly code.

Related information

–

--signed-bitfields

Command line syntax

--signed-bitfields

Description

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

Example

```
cpcp --signed-bitfields test.c
```

Related information


-

--static

Command line syntax

--static

Description

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

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

Example

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

Related information

-

–t (––tradeoff)

Command line syntax

–t{**0**|**1**|**2**|**3**|**4**}

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

Default

–t2

Description

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

Default the compiler balances speed and size while optimizing (**–t2**).

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

Example

To set the trade-off level for the used optimizations:

```
cpcp –t4 test.c
cpcp ––tradeoff=4 test.c
```

The compiler uses the default medium optimization level and optimizes for code size rather than for speed.

Related information

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

-U (--undefine)

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.

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__`:

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

Related information

Compiler option **-D** (Define macro)

Section 2.7, *Predefined Macros*, in Chapter *Using the Compiler* of the *User's Manual*.

-u (--uchar)

Command line syntax

-u

--uchar

Description

Treat 'character' type variables as 'unsigned character' variables. By default `char` is the same as specifying `signed char`. With **-u** `char` is the same as `unsigned char`.

Example

With the following command `char` is treated as `unsigned char`:

```
cpcp -u test.c
cpcp --uchar test.c
```

Related information

-

-V (--version)

Command line syntax

-V

--version

Description

Display version information. The compiler ignores all other options or input files.

Example

```
cpcp -v
cpcp --version
```

The compiler does not compile any files but displays the following version information:

```
 PCP VX-toolset C compiler        vxx.yrz Build 000
Copyright 2006-year Altium BV            Serial# 00000000
```

Related information

-

-w (--no-warnings)

Command line syntax

-w[*nr*]

--no-warnings[**=***nr*]

Description

With this option you can suppresses all warning messages or specific warning messages.

- If you do not specify this option, all warnings are reported.
- If you specify this option but without numbers, all warnings are suppressed.
- If you specify this option with a number, only the specified warning is suppressed. You can specify the option **-w** multiple times.

Example

To suppress all warnings:

```
cpcp test.c -w
cpcp test.c --no-warnings
```

To suppress warnings 135 and 136:

```
cpcp test.c -w135 -w136
cpcp test.c --no-warnings=135 --no-warnings=136
```

Related information

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

--warnings-as-errors

Command line syntax

--warnings-as-errors[=*number*,...]

Description

If the compiler encounters an error, it stops compiling. When you use this option without arguments, you tell the compiler to treat all warnings as errors. This means that the exit status of the compiler will be non-zero after one or more compiler warnings. As a consequence, the compiler now also stops after encountering a warning.

You can also limit this option to specific warnings by specifying a comma-separated list of warning numbers.

Example

```
cpcp --warnings-as-errors test.c
```

When a warning occurs, the compiler considers it as an error.

Related information

Compiler option **-w** (suppress some or all warnings)

5.2 ASSEMBLER OPTIONS

This section lists all assembler options.

Short and long option names

Options have both short and long names. Short option names always begin with a single minus (**-**) character, long option names always begin with two minus (**--**) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

Options can have flags or suboptions. To switch a flag 'on', use a lowercase letter or a *+longflag*. To switch a flag off, use an uppercase letter or a *-longflag*. Separate *longflags* with commas. The following two invocations are equivalent:

```
aspcp -Lmx test.src
aspcp --list-format=+macro,+macro-expansion test.src
```

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

–? (––help)

Command line syntax

–?

––help[**=options**]

Description

Displays an overview of all command line options. When you specify the **options** argument, a list with option descriptions is displayed.

Example

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

```
aspcp -?
aspcp --help
aspcp
```

The following invocation displays extended information about all options:

```
aspcp --help=options
```

Related information

-

-C (--cpu)

Command line syntax

-C*cpu*

--cpu=*cpu*

Description

With this option you define the target processor for which you create your application. Make sure you choose a target processor *with* pcp!

The assembler automatically includes the register file `reg`*`cpu`*`.def`, unless you specify assembler option **--no-tasking-sfr**.

Example

To define this on the command line:

```
aspcp -Ctc1920b test.src
aspcp --cpu=tc1920b test.src
```

The assembler assembles `test.src` for the TC11IB processor and includes the register file `reg1920b.def`.

To avoid conflicts, make sure you specify the same target processor as you did for the compiler.

Related information

Assembler option **--no-tasking-sfr** (Do not include .def file)

Compiler option **-C** (Use SFR definitions for CPU)
Control program option **-C** (Use SFR definitions for CPU)

Section 5.4, *Calling the Assembler*, in Chapter *Using the Assembler* of the *User's Manual*.

–c (––case-insensitive)

Command line syntax

–c

––case–insensitive

Description

With this option you tell the assembler not to distinguish between upper and lower case characters. By default the assembler considers upper and lower case characters in labels and user-defined symbols as different characters. Note that instruction mnemonics, register names, directives and controls are always treated case insensitive.

Assembly source files that are generated by the compiler must always be assembled case sensitive. When you are writing your own assembly code, you may want to specify the case insensitive mode.

Example

To assemble case insensitive:

```
aspcp –c test.src
aspcp ––case–insensitive test.src
```

The assembler considers upper and lower case characters as being the same. So, for example, the label `LabelName` is the same label as `labelname`.

Related information

Linker option **––case–sensitive** (Link case insensitive)

--check

Command line syntax

--check

Description

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

The assembler reports any warnings and/or errors.

Example

To check for syntax errors, without generating code:

```
aspcp --check test.src
```

Related information

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

–D (––define)

Command line syntax

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

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

Description

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

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

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

This option has the same effect as defining symbols via the `.SET`, and `.EQU` directives. (similar to `#define` in the C language). With the `.MACRO` directive you can define more complex macros.

Example

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

```
.IF DEMO == 1
    ...         ; instructions for demo application
.ELSE
    ...         ; instructions for the real application
.ENDIF
```

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

```
aspcp -DDEMO test.src
aspcp -DDEMO=1 test.src

aspcp --define=DEMO test.src
aspcp --define=DEMO=1 test.src
```

Note that all four invocations have the same effect.

Related information

Assembler option **–f** (Specify an option file)

Section 3.10.5, *Conditional Assembly*, in Chapter *PCP Assembly Language* of the *User's Manual*.

--diag

Command line syntax

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

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

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

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to `stdout` (normally your screen) and in the format you specify.

To create a file with the descriptions, you must redirect the output.

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

With this option the assembler does not assemble any files.

Example

To display an explanation of message number 241, enter:

```
aspcp --diag=241
```

This results in the following message and explanation:

```
W241: additional input files will be ignored

The assembler supports only a single input file. All
other input files are ignored.
```

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

```
aspcp --diag=html:all > aserrors.html
```

Related information

Section 5.8, *Assembler Error Messages*, in Chapter *Using the Assembler* of the *User's Manual*.

–E (––preprocess)

Command line syntax

–E

––preprocess

Description

With this option the assembler will only preprocess the assembly source file. The assembler sends the preprocessed file to stdout.

Example

```
aspcp –E test.src

aspcp ––preprocess test.src
```

Related information


–

--emit-locals

Command line syntax

--emit-locals[=*flag*,...]

You can set the following flags (when you specify no flags, the default is **Es**):

e/**E**	(**+**/**-equs**)	emit local EQU symbols
s/**S**	(**+**/**-symbols**)	emit local non-EQU symbols

Description

With the option **--emit-locals=+equs** the assembler also emits local EQU symbols to the object file. Without a flag or with **--emit-locals=+symbols** the assembler also emits non-EQU local symbols to the object file. Normally, only global symbols are emitted. Having local symbols in the object file can be useful for debugging.

Example

To emit local (non-EQU) symbols, enter:

```
aspcp --emit-locals test.src
```

Related information

-

--error-file

Command line syntax

--error-file[=*file*]

Description

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

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

Example

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

```
aspcp --error-file=errors.ers test.src
```

Related information

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

--error-limit

Command line syntax

--error-limit=*number*

Description

With this option you tell the assembler to only emit the specified maximum number of errors. When 0 (null) is specified, the assembler emits all errors. Without this option the maximum number of errors is 42.

Example

To stop assembling after 10 errors occurred, enter:

```
aspcp --error-limit=10 test.src
```

Related information


-

–f (––option-file)

Command line syntax

–f *file*,...

––option–file=*file*,...

Description

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

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

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

Format of an option file

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

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

'This has a double quote " embedded'

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

  Note that adjacent strings are concatenated.

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

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

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

Example

Suppose the file `myoptions` contains the following lines:

```
-Ctc1920b
test.src
```

Specify the option file to the assembler:

```
aspcp -f myoptions
aspcp --option-file=myoptions
```

This is equivalent to the following command line:

```
aspcp -Ctc1920b test.src
```

Related information

-

–g (––debug-info)

Command line syntax

–g[*flag*]

––debug–info[**=***flag*]

You can set the following flags:

a/**A**	(**+**/**–asm**)	Assembly source line information
h/**H**	(**+**/**–hll**)	Pass HLL debug information
l/**L**	(**+**/**–local**)	Local symbols debug information
s/**S**	(**+**/**–smart**)	Smart debug information

Default

–gs

Description

With this option you tell the assembler to generate debug information. If you do not use this option or if you specify **–g** without any flags, the default is **–gs**.

You cannot specify **–gah**. Either the assembler generates assembly source line information, or it passes HLL debug information.

When you specify **–gs**, the assembler selects which flags to use. If high level language information is available in the source file, the assembler passes this information (same as **–gAhL**). If not, the assembler generates assembly source line information and local symbols debug information (same as **–gaHl**).

With **–gAHLS** the assembler does not generate any debug information.

Example

To disable symbolic debug information, turn all flags off:

```
aspcp -gAHLS test.src
aspcp --debug-info=-asm,-hll,-local,-smart test.src
```

To enable smart debugging, enter:

```
aspcp -gs test.src
aspcp --debug-info=+smart test.src
```

Related information

-

–H (––include-file)

Command line syntax

–H*file*,...

––include-file=*file*,...

Description

With this option you include one extra file at the beginning of the assembly source file, before other includes. This is the same as specifying `.INCLUDE 'file'` at the beginning of your assembly sources.

Example

```
aspcp -Hmyinc.inc test1.src
aspcp --include-file=myinc.inc test1.src
```

The file `myinc.inc` is included at the beginning of `test1.src` before it is assembled.

Related information

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

Section 5.6, *How the Assembler Searches Include Files*, in Chapter *Using the Assembler* of the *User's Manual*.

-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.

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

1. The absolute pathname, if specified in the `.INCLUDE` directive. Or, if no path or a relative path is specified, the same directory as the source file.
2. The path that is specified with this option.
3. The path that is specified in the environment variable ASPCPINC 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:

```
aspcp -Ic:\proj\include test.src
aspcp --include-directory=c:\proj\include test.src
```

First the assembler looks in the directory where `test.src` is located for the file `myinc.inc`. If it does not find the file, it looks in the directory `c:\proj\include` for the file `myinc.inc` (this option).

Related information

Section 5.6, *How the Assembler Searches Include Files*, in Chapter *Using the Assembler* of the *User's Manual*.

Section 1.3.1, *Configuring the Command Line Environment*, in Chapter *Software Installation* of the *User's Manual*.

Assembler option **–H** (Include file at the start of the input files)

-i (--symbol-scope)

Command line syntax

-i{g|l}

--symbol-scope={global|local}

Default

-il

Description

With this option you tell the assembler how to treat symbols that you have not specified explicitly as global or local.

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

Example

```
aspcp -ig test.src
aspcp --symbol-scope=global test.src
```

The assembler treats all symbols as global symbols unless they are defined as local symbols in the assembly source file.

Related information

-

–k (––keep-output-files)

Command line syntax

–k

––keep–output–files

Description

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

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

Use this option when you still want to use the generated object. For example when you know that a particular error does not result in a corrupt object file.

Example

```
aspcp –k test.src
```

When an error occurs during assembly, the generated output file `test.o` will *not* be removed.

Related information

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

-L (--list-format)

Command line syntax

-L*flags*

--list-format=*flags*

You can set the following flags:

0		same as **-LCDEGILMNPQRVWXY**
1		same as **-Lcdegilmnpqrvwxy**
d/**D**	(**+**/**-section**)	Section directives
e/**E**	(**+**/**-symbol**)	Symbol definition directives
g/**G**	(**+**/**-generic-expansion**)	Generic instruction expansion
i/**I**	(**+**/**-generic**)	Generic instructions
m/**M**	(**+**/**-macro**)	Macro definitions
n/**N**	(**+**/**-empty-line**)	Empty source lines
p/**P**	(**+**/**-conditional**)	Conditional assembly
q/**Q**	(**+**/**-equate**)	Assembler .EQU and .SET directives
r/**R**	(**+**/**-relocations**)	Relocation characters ('r')
v/**V**	(**+**/**-equate-values**)	Assembler .EQU and .SET values
w/**W**	(**+**/**-wrap-lines**)	Wrapped source lines
x/**X**	(**+**/**-macro-expansion**)	Macro expansions
y/**Y**	(**+**/**-cycle-count**)	Cycle counts

Default

-LDEGiMnPqrVWXy

Description

With this option you specify which information you want to include in the list file. Use this option in combination with the option **-l** (**--list-file**).

If you do not specify this option, the assembler uses the default: **-LDEGiMnPqrVWXy**.

With option **-tl**, the assembler also writes section information to the list file.

Example

```
aspcp -l -Ldm test.src
aspcp --list-file --list-format=+section,+macro
     test.src
```

The assembler generates a list file that includes all default information plus section directives and macro definitions.

Related information

Assembler option **–l** (Generate list file)
Assembler option **–tl** (Display section information in list file)

Linker option **–M** (Generate map file)

Section 6.1, *Assembler List File Format*, in Chapter *List File Formats*.

-l (--list-file)

Command line syntax

-l

--list-file

Description

With this option you tell the assembler to generate a list file. A list file shows the generated object code and the relative addresses. Note that the assembler generates a relocatable object file with relative addresses.

Example

To generate a list file with the name `test.lst`, enter:

```
aspcp -l test.src
aspcp --list-file test.src
```

Related information

Assembler option **-L** (List file formatting options)

Linker option **-M** (Generate map file)

Section 6.1, *Assembler List File Format*, in Chapter *List File Formats*.

-m (--preprocessor-type)

Command line syntax

-m{n|t}

--preprocessor-type={none|tasking}

Default

-mt

Description

With this option you select the preprocessor that the assembler will use. By default, the assembler uses the TASKING preprocessor.

When the assembly source file does not contain any preprocessor symbols, you can specify the assembler not to use a preprocessor.

Example

```
aspcp test.src
aspcp -mt test.src
aspcp --preprocessor=tasking test.src
```

These invocations have the same effect: the assembler preprocesses the file `test.src` with the TASKING preprocessor.

Related information


-

--no-tasking-sfr

Command line syntax

--no-tasking-sfr

Description

Normally, the assembler includes a special function register (SFR) file before compiling. The assembler automatically selects the SFR file belonging to the target you select on the **Processor definition** page of the Processor options (assembler option **-C**).

With this option the assembler does not include the register file `reg`*`cpu`*`.def` as based on the selected target processor.

Use this option if you want to use your own set of SFR '.def' files.

Example

```
aspcp -Ctc1920b --no-tasking-sfr test.src
```

The register file `regtc1920b.def` is not included, but the assembler allows the use of MMU instructions due to **-C**.

Related information

Assembler option **-C** (Select CPU)

–O (––optimize)

Command line syntax

–O*flags*

––optimize=*flags*

You can set the following flags:

g/**G**	(**+**/**–generics**)	Allow generic instructions
s/**S**	(**+**/**–instr–size**)	Optimize instruction size

Default

–Ogs

Description

With this option you can control the level of optimization. If you do not use this option, **–Ogs** is the default.

Example

The following invocations are equivalent and result all in the default optimizations:

```
aspcp test.src
aspcp -Ogs test.src
aspcp --optimize=+generics,+instr-size test.src
```

Related information

Section 5.3, *Assembler Optimizations*, in Chapter *Using the Assembler* of the *User's Manual*.

-o (--output)

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 `.o`.

Example

```
aspcp -o relobj.o asm.src
aspcp --output=relobj.o asm.src
```

The assembler creates the file `relobj.o` for the assembled file `asm.src`.

Without the option **-o**, the assembler uses the name of the input file and creates `asm.o`.

Related information

-

–P (––prefix)

Command line syntax

–P*prefix*

––prefix=*prefix*

Description

PCP assembler only. With this option you can specify a prefix to use for global and external symbols. The prefix can be useful to disinguish these symbols from symbols generated by the **aspcp** assembler.

Example

The add the prefix _PCP_ to global/extenal symbols, enter:

```
aspcp -P_PCP_ test.pcp
aspcp --prefix=_PCP_ test.pcp
```

Related information


-

-p (--pcptype)

Command line syntax

-p*pcptype*

--pcptype=*pcptype*

You can specify the following PCP types:

1	PCP 1 syntax
2	PCP 2 syntax
tc1775	generate code for the PCP on the TC1775

Description

PCP assembler only. With this option you can choose the syntax for the PCP assembler.

Example

The select to generate code for the PCP on the TC1775, enter:

```
aspcp -ptc1775 test.pcp
aspcp --pcptype=tc1775 test.pcp
```

Related information


-

–t (––section-info)

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

```
aspcp -l -tcl test.src
aspcp -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`.

```
Section summary:

  REL                    4   .zbss_clr_test1
  REL                   46   .text_test1
  REL                    4   .zdata_rom_test1
```

Related information

Assembler option **–l** (Generate list file)

-V (--version)

Command line syntax

-V

--version

Description

Display version information. The assembler ignores all other options or input files.

Example

```
aspcp -V
aspcp --version
```

The assembler does not assemble any files but displays the following version information:

```
TASKING PCP VX-toolset Assembler       vxx.yrz Build nnn
Copyright years Altium BV              Serial# 00000000
```

Related information

-

–w (––no-warnings)

Command line syntax

–w[*nr*,...]

––no-warnings[**=***nr*,...]

Description

With this option you can suppresses all warning messages or specific warning messages.

- If you do not specify this option, all warnings are reported.
- If you specify this option but without numbers, all warnings are suppressed.
- If you specify this option with a number, only the specified warning is suppressed. You can specify the option **–w** multiple times.

Example

To suppress all warnings:

```
aspcp -w test.src
aspcp --no-warnings test.src
```

To suppress warnings 135 and 136:

```
aspcp -w135,136 test.src
aspcp --no-warnings=135,136 test.src
```

Related information

Assembler option **––warnings-as-errors** (Treat warnings as errors)

--warnings-as-errors

Command line syntax

--warnings-as-errors[**=***number*,...]

Description

If the assembler encounters an error, it stops assembling. When you use this option without arguments, you tell the assembler to treat all warnings as errors. This means that the exit status of the assembler will be non-zero after one or more assembler warnings. As a consequence, the assembler now also stops after encountering a warning.

You can also limit this option to specific warnings by specifying a comma-separated list of warning numbers.

Example

```
aspcp --warnings-as-errors test.src
```

When a warning occurs, the assembler considers it as an error. No object file is generated, unless you specify option **-k** (**--keep-output-files**).

Related information

Assembler option **-w** (suppress some or all warnings)

5.3 LINKER OPTIONS

See section 5.4, *Control Program Options*.

Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (**-**) character, long option names always begin with two minus (**--**) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

Options can have flags or suboptions. To switch a flag 'on', use a lowercase letter or a *+longflag*. To switch a flag off, use an uppercase letter or a *-longflag*. Separate *longflags* with commas. The following two invocations are equivalent:

```
lpcp -mfkl test.o
lpcp --map-file-format=+files,+link,+locate test.o
```

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

–? (––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:

```
lpcp -?
lpcp --help
lpcp
```

To see a detailed description of the available options, enter:

```
lpcp --help=options
```

Related information


-

-c (--chip-output)

Command line syntax

-c[*basename*]:*format*[**:***addr_size*]**,**...

--chip-output=[*basename*]:*format*[:*addr_size*]**,**...

You can specify the following formats:

IHEX Intel Hex
SREC Motorola S-records

The *addr_size* specifies the size of the addresses in bytes (record length). For Intel Hex you can use the values: **1**, **2** and **4** (default). For Motorola S you can specify: **2** (S1 records), **3** (S2 records, default) or **4** bytes (S3 records).

Description

With this option you specify the Intel Hex or Motorola S-record output format for loading into a PROM-programmer. The linker generates a file for each ROM memory defined in the LSL file, where sections are located:

```
memory memname
{  type=rom;  }
```

The name of the file is the name of the memory device that was emitted with extension `.hex` or `.sre`. Optionally you can specify a *basename* which prepends the generated file name.

Examples

To generate Intel Hex output files for each defined memory, enter the following on the command line:

```
p -cmyfile:IHEX test1.o
lpcp --chip-output=myfile:IHEX test1.o
```

This generates the file `myfile_memname.hex`.

Related information

Linker option **-o** (output file),
Section 7.2, *Motorola S-Record Format*,
Section 7.3, *Intel Hex Record Format*, in Chapter *Object File Formats*.

--case-insensitive

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.

Assembly source files that are generated by the compiler must always be assembled and thus linked case sensitive. When you have written your own assembly code and specified to assemble it case insensitive, you must also link the `.o` file case insensitive.

Example

To link case insensitive:

```
lpcp --case-insensitive test.o
```

The linker considers upper and lower case characters as being the same. So, for example, the label `LabelName` is considered the same label as `labelname`.

Related information

Assembler option **-c** (Assemble case insensitive)

–D (––define)

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

Define *macro* to the preprocessor, as in #define. Any number of symbols can be defined. The definition can be tested by the preprocessor with `#if`, `#ifdef` and `#ifndef`, for conditional locating.

Example

To define the RESET vector, interrupt table start address and trap table start address which is used in the linker script file `tc1v1_3.lsl`, enter:

```
lpcp test.o -otest.elf -dtc1v1_3.lsl
-DRESET=0xa0000000
     -DINTTAB=0xa00f0000 --define=TRAPTAB=0xa00f2000
```

Related information

Linker option **–f** (Name of invocation file)

-d (--lsl-file)

Command line syntax

-d*file*
--lsl-file=*file*

Description

With this option you specify a linker script file to the linker. If you do not specify this option, the linker uses a default script file. You can specify the existing file `target.lsl` or the name of a manually created linker script file. You can use this option multiple times. The linker processes the LSL files in the order in which they appear on the command line.

The linker script file contains vital information about the core for the locating phase of the linker. A linker script file is coded in LSL and contains the following types of information:

- the architecture definition describes the core's hardware architecture.
- the memory definition describes the physical memory in the system.
- the section layout definition describes how to locate sections in memory.

Example

To read linker script file information from file `tc1v1_3.lsl`:

```
lpcp -dtc1v1_3.lsl test.o
lpcp --lsl-file=tc1v1_3.lsl test.o
```

Related information

Linker option **--lsl-check** (Check LSL file(s) and exit)

Section 6.7, *Controlling the Linker with a Script* in Chapter *Linker* of the *User's Manual*.

--diag

Command line syntax

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

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

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

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to `stdout` (normally your screen) and in the format you specify.

To create a file with the descriptions, you must redirect the output.

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

With this option the linker does not link any files.

Example

To display an explanation of message number 106, enter:

```
lpcp --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:

```
lpcp --diag=html:all > lerrors.html
```

Related information

Section 6.10, *Linker Error Messages*, in Chapter *Using the Linker* of the *User's Manual*.

–e (––extern)

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:

```
lpcp mylib.a
```

Nothing is linked and no output file will be produced, because there are no unresolved symbols when the linker searches through `mylib.a`.

```
lpcp -e _START mylib.a
lpcp --extern=_START mylib.a
```

In this case the linker searches for the symbol `_START` in the library and (if found) extracts the object that contains `_START`, the startup code. If this module contains new unresolved symbols, the linker looks again in `mylib.a`. This process repeats until no new unresolved symbols are found.

Related information

Section 6.4.1, *Specifying Libraries to the Linker*, in Chapter *Using the Linker* of the *User's Manual*.

--error-file

Command line syntax

--error-file[**=***file*]

Description

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

If you do not specify a filename, the error file is `lpcp.elk`.

Example

```
lpcp --error-file=my.elk test.o
```

The linker writes error messages to the file `my.elk` instead of `stderr`.

Related information

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

--error-limit

Command line syntax

--error-limit=*number*

Description

With this option you tell the linker to only emit the specified maximum number of errors. When 0 (null) is specified, the linker emits all errors. Without this option the maximum number of errors is 42.

Example

To stop linking after 10 errors occurred, enter:

```
lpcp --error-limit=10 test.o
```

Related information

 -

-f (--option-file)

Command line syntax

-f *file*,...
--option-file=*file*,...

Description

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

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

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

Format of an option file

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

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

'This has a double quote " embedded'

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

  Note that adjacent strings are concatenated.

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

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

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

Example

Suppose the file `myoptions` contains the following lines:

```
-Mmymap          (generate a map file)
test.o           (input file)
-Lc:\mylibs      (additional search path for system libraries)
```

Specify the option file to the linker:

```
lpcp -f myoptions
lpcp --option-file=myoptions
```

This is equivalent to the following command line:

```
lpcp -Mmymap test.o -Lc:\mylibs
```

Related information

–

--first-library first

Command line syntax

--first-library-first

Description

When the linker processes a library it searches for symbols that are referenced by the objects and libraries processed so far. If the library contains a definition for an unresolved reference the linker extracts the object that contains the definition from the library.

By default the linker processes object files and libraries in the order in which they appear on the command line. If you specify the option **--first-library-first** the linker always tries to take the symbol definition from the library that appears first on the command line before scanning subsequent libraries.

This is for example useful when you are working with a newer version of a library that partially overlaps the older version. Because they do not contain exactly the same functions, you have to link them both. However, when a function is present in both libraries, you may want the linker to extract the most recent function.

Example

```
lpcp --first-library-first a.a test.o b.a
```

If the file `test.o` calls a function which is both present in `a.a` and `b.a`, normally the function in `b.a` would be extracted. With this option the linker first tries to extract the symbol from the first library `a.a`.

Note that routines in `b.a` that call other routines that are present in both `a.a` and `b.a` are now also resolved from `a.a`.

Related information

Linker option **--no-rescan** (Do not rescan libraries)

–I (––include-directory)

Command line syntax

–I*path*,...

––include–directory=*path*,...

Description

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

The order in which the linker searches for LSL include files is:

1. The pathname in the LSL file and the directory where the LSL files are located (only for #include files that are enclosed in "")
2. The path that is specified with this option.
3. The default `$(PRODDIR)\include.lsl` directory relative to the installation directory.

Example

Suppose that the LSL file `lslfile.lsl` contains the following lines:

```
#include "mypart.lsl"
```

You can call the linker as follows:

```
lpcp -Imyinclude -dlslfile.lsl test.o

lpcp --include-directory=myinclude
    --lsl-file=lslfile.lsl test.o
```

First the linker looks in the directory where `lslfile.lsl` is located for the file `mypart.lsl`. If it does not find the file, it looks in `myinclude` subdirectory relative to the current directory for the file `mypart.lsl` (this option). Finally it looks in the directory `$(PRODDIR)\include.lsl`.

Related information

Linker option **–d** (Linker script file)

-i (--user-provided-initialization-code)

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.

Example

To link with your own startup code:

```
lpcp -i test.o
lpcp --user-provided-initialization-code test.o
```

Related information

-

-k (--keep-output-files)

Command line syntax

-k
--keep-output-files

Description

If an error occurs during linking, the resulting output file may be incomplete or incorrect. With this option you keep the generated output files when an error occurs.

By default the linker removes the generated output files when an error occurs. This is useful when you use the make utility **mkpcp**. If the erroneous files are not removed, the make utility may process corrupt files on a subsequent invocation.

Use this option when you still want to use the generated file. For example when you know that the error(s) do not result in a corrupt output file, or when you want to inspect the output file, or send it to Altium support.

Example

```
lpcp -k test.o
lpcp --keep-output-files test.o
```

When an error occurs during linking, the generated output file `test.elf` will *not* be removed.

Related information


-

–L (––library-directory / ––ignore-default-library-path)

Command line syntax

–L*path*,...
––library–directory=*path*,...

–L
––ignore–default–library–path

Description

With this option you can specify the path(s) where your system libraries, specified with the **–l** option, are located. If you want to specify multiple paths, use the option **–L** for each separate path.

The default path is `$(PRODDIR)\lib`.

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 LIBTC1V1_2, LIBTC1V1_3 or LIBTC2. So, the linker ignores steps 2, 3 and 4 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 LIBTC1V1_2, LIBTC1V1_3 or LIBTC2 when the product was installed.
3. The default directory `$(PRODDIR)\lib`.
4. The processor specific directory, for example `$(PRODDIR)\lib\pcp1`.

Example

Suppose you call the linker as follows:

```
lpcp test.o -Lc:\mylibs -lc
```

First the linker looks in the directory `c:\mylibs` for library `libc.a` (this option).

If it does not find the requested libraries, it looks in the directory that is set with the environment variable LIBTC1V1_2, LIBTC1V1_3 or LIBTC2.

Then the linker looks in the default directory `$(PRODDIR)\lib` for libraries.

Related information

Linker option **–l** (Link system library)

Section 6.4.2, *How the Linker Searches Libraries*, in Chapter *Using the Linker* of the *User's Manual*.

Section 1.3.1, *Configuring the Command Line Environment*, in Chapter *Software Installation* of the *User's Manual*.

-l (--library)

Command line syntax

-l*name*
--library=*name*

Description

With this option you tell the linker to search also in system library `lib`*`name`*`.a`, where *name* is a string. The linker first searches for system libraries in any directories specified with **-L***path*, then in the directories specified with the environment variable LIBTC1V1_2, LIBTC1V1_3 or LIBTC2, unless you used the option **-L** without a directory.

If you use the `libc.a` library, you must always link the `libfp.a` library as well. Remember that the order of the specified libraries is important!

Example

To search in the system library `libfp.a` (floating-point library):

```
lpcp test.o mylib.a -lfp
lpcp test.o mylib.a --library=fp
```

The linker links the file `test.o` and first looks in `mylib.a` (in the current directory only), then in the system library `libfp.a` to resolve unresolved symbols.

Related information

Linker option **-L** (Additional search path for system libraries)

Section 6.4.1, *Specifying Libraries to the Linker*, in Chapter *Using the Linker* of the *User's Manual*.

--link-only

Command line syntax

--link-only

Description

With this option you suppress the locating phase. The linker stops after linking. The linker complains if any unresolved references are left.

Example

```
lpcp --link-only hello.o
```

The linker checks for unresolved symbols and creates the file `task1.out`.

Related information

Control program option **-cl** (Stop after linking)

--lsl-check

Command line syntax

--lsl-check

Description

With this option the linker just checks the syntax of the LSL file(s) and exits. No linking or locating is performed.

Example

To check the LSL file(s) and exit:

```
lpcp --lsl-check --lsl-file=mylslfile.lsl
```

Related information

Linker option **-d** (Linker script file)

Linker option **--lsl-dump** (Dump LSL info)

Chapter 8, *Linker Script Language*.

--lsl-dump

Command line syntax

--lsl-dump[=*file*]

Description

With this option you tell the linker to dump the LSL part of the map file in a separate file, independent of the **-M** (generate map file) option. If you do not specify a filename, the file `lpcp.ldf` is used.

Example

```
lpcp --lsl-dump=mydump.ldf test.o
```

The linker dumps the processor and memory info from the LSL file in the file `mydump.ldf`.

Related information

Linker option **-m** (Map file formatting options)

-M (--map-file)

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 specified 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`.

A linker map file is a text file that shows how the linker has mapped the sections and symbols from the various object files (`.o`) to the linked object file. A locate part shows the absolute position of each section. External symbols are listed per space with their absolute address, both sorted on symbol and sorted on address.

With the option **-m** (map file formatting) you can specify which parts you want to place in the map file.

Example

To generate a map file (`test.map`):

```
lpcp -Mtest.map test.o
lpcp --map-file=test.map test.o
```

The control program by default tells the linker to generate a map file.

Related information

Linker option **-m** (Map file formatting options)

Section 6.2, *Linker Map File Format*, in Chapter *List File Formats*.

–m (––map-file-format)

Command line syntax

–m*flags*

––map–file–format=*flags*

You can set the following flags:

0	same as **–mcfikLMNoQrSU**	(link info)
1	same as **–mCfiKlMNoQRSU**	(locate info)
2	same as **–mcfiklmNoQrSu**	(most info)

c/**C**	(**+**/**–callgraph**)	Call graph info
f/**F**	(**+**/**–files**)	Processed files info
i/**I**	(**+**/**–invocation**)	Invocation and tool info
k/**K**	(**+**/**–link**)	Link result info
l/**L**	(**+**/**–locate**)	Locate result info
m/**M**	(**+**/**–memory**)	Memory usage info
n/**N**	(**+**/**–nonalloc**)	Non alloc info
o/**O**	(**+**/**–overlay**)	Overlay info
q/**Q**	(**+**/**–statics**)	Module local symbols
r/**R**	(**+**/**–crossref**)	Cross references info
s/**S**	(**+**/**–lsl**)	Processor and memory info
u/**U**	(**+**/**–rules**)	Locate rules

Default

–m2

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**

Example

```
lpcp –Mtest.map –mF test.o
lpcp ––map–file=test.map
    ––map–file–format=–files test.o
```

The linker generates the map file `test.map` that includes all default information, but not the processed files part.

Related information

Linker option **–M** (Generate map file)

Section 6.2, *Linker Map File Format*, in Chapter *List File Formats*.

--misra-c-report

Command line syntax

--misra-c-report[=*file*]

Description

With this option you tell the linker to create a MISRA-C Quality Assurance report. This report lists the various modules in the project with the respective MISRA-C settings at the time of compilation. If you do not specify a filename, the file *name*`.mcr` is used.

Example

```
lpcp --misra-c-report test.o
```

The linker creates a MISRA-C report file `test.mcr`.

Related information

Compiler option **--misrac**

--munch

Command line syntax

--munch

Description

With this option you tell the linker to activate the muncher in the pre-locate phase.

The data sections are initialized when the application is downloaded. The data sections are not re-initialized when the application is restarted.

Example

```
lpcp --munch test.o
```

The linker activates the muncher in the pre-locate phase while linking the file `test.o`.

Related information

-

-N (--no-rom-copy)

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 sections. However, no entries to copy data sections from ROM to RAM are placed in the copy table.

The data sections are initialized when the application is downloaded. The data sections are not re-initialized when the application is restarted.

Example

```
lpcp -N test.o
lpcp --no-rom-copy test.o
```

The linker does not generate ROM copies for data sections.

Related information


-

--no-rescan

Command line syntax

--no-rescan

Description

When the linker processes a library, it searches for symbol definitions that are referenced by the objects and libraries processed so far. If the library contains a definition for an unresolved reference, the linker extracts the object that contains the definition from the library. The linker processes object files and libraries in the order in which they appear on the command line.

When all objects and libraries are processed the linker checks if there are unresolved symbols left. If so, the default behavior of the linker is to rescan all libraries in the order given on the command line. The linker stops rescanning the libraries when all symbols are resolved, or when the linker could not resolve any symbol(s) during the rescan of all libraries. Notice that resolving one symbol may introduce new unresolved symbols.

With this option, you tell the linker to scan the object files and libraries only once. When the linker has not resolved all symbols after the first scan, it reports which symbols are still unresolved. This option is useful if you are building your own libraries. The libraries are most efficiently organized if the linker needs only one pass to resolve all symbols.

Example

To scan the libraries only once:

```
lpcp --no-rescan test.o a.a b.a
```

The linker resolves all unresolved symbols while scanning the object files and libraries and reports all remaining unresolved symbols after this scan.

Related information

Linker option **--first-library-first** (Scan libraries in the specified order)

--non-romable

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.

Example

```
lpcp --non-romable test.o
```

The linker locates all ROM sections in RAM.

Related information


-

–O (––optimize)

Command line syntax

–O*flags*

––optimize=*flags*

You can set the following flags:

c/**C** (**+**/**–delete–unreferenced–sections**)
Delete unreferenced sections from the output file
(no effect on sources compiled with debug information)

l/**L** (**+**/**–first–fit–decreasing**)
Use a 'first fit decreasing' algorithm to locate
unrestricted sections in memory.

t/**T** (**+**/**–copytable–compression**)
Locate (unrestricted) sections in such a way that
the size of the copy table is as small as possible.

x/**X** (**+**/**–delete–duplicate–code**)
Delete duplicate code sections from the output file

y/**Y** (**+**/**–delete–duplicate–data**)
Delete duplicate constant data sections from the output file

Use the following options for predefined sets of flags:

–O0	(**––optimize=0**)	No optimization. Alias for: **–OCLTXY**
–O1	(**––optimize=1**)	Normal optimization (default). Alias for: **–OcLtxy**
–O2	(**––optimize=2**)	All optimizations. Alias for: **–Ocltxy**

Default

–O1

Description

With this option you can control the level of optimization. If you do not use this option, **–OcLtxy** (**–O1**) is the default.

Example

The following invocations are equivalent and result all in the default optimizations.

```
lpcp test.o
lpcp -O test.o
lpcp -O1 test.o
lpcp -OcLtxy test.o

lpcp --optimize test.o
lpcp --optimize=1 test.o
lpcp --optimize=-delete-unreferenced-sections,
  +first-fit-decreasing,-copytable-compression,
  -delete-duplicate-code,-delete-duplicate-data test.o
```

Related information

Section 6.2.3, *Linker Optimizations*, in Chapter *Using the Linker* of the *User's Manual*.

-o (--output)

Command line syntax

-o[*filename*][**:***format*[**:***addr_size*][**,***space_name*]]...

--output=[*filename*][**:***format*[**:***addr_size*][**,***space_name*]]...

You can specify the following formats:

ELF	ELF/DWARF
IEEE	IEEE-695
IHEX	Intel Hex
SREC	Motorola S-records

Description

By default, the linker generates an output file in ELF/DWARF format, with the name `task1.elf`.

With this option you can specify an alternative *filename*, and an alternative output *format*. The default output format is the format of the first input file.

You can use the **-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 `taskn`.

IHEX and SREC formats

If you specify the Intel Hex format or the Motorola S-records format, you can use the argument *addr_size* to specify the size of addresses in bytes (record length). For Intel Hex you can use the values: **1**, **2**, and **4** (default). For Motorola S-records you can specify: **2** (S1 records), **3** (S2 records, default) or **4** bytes (S3 records).

With the argument *space_name* you can specify the name of the address space. The name of the output file will be *filename* with the extension `.hex` or `.sre` and contains the code and data allocated in the specified space. The other address spaces are also emitted whereas there output files are named *filename_spacename* with the extension `.hex` or `.sre`.

If you do not specify *space_name*, or you specify a non-existing space, the default address space is filed in.

Use option **-c** (**--chip-output**) to create Intel Hex or Motorola S-record output files for each chip (suitable for loading into a PROM-programmer).

Example

To create the output file `myfile.hex` of the address space named `linear`:

```
lpcp test.o -omyfile.hex:IHEX:2,linear
lpcp test.o --output=myfile.hex:IHEX:2,linear
```

Related information

Linker option **-c** (Generate an output file for each chip)
Section 7.1, *ELF/DWARF Object Format*, in Chapter *Object File Formats*.

-r (--incremental)

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.o`, `test2.o` and `test3.o` are incrementally linked:

1. `lpcp -r test1.o test2.o -otest.out`

   `test1.o` and `test2.o` are linked

2. `lpcp --incremental test3.o test.out`

   `test3.o` is linked together with `test.out`. File `task1.out` is created.

3. `lpcp task1.out` (*`task1.out` is located*)

Related information

Section 6.5, *Incremental Linking*, in Chapter *Using the Linker* of the *User's Manual*.

–S (––strip-debug)

Command line syntax

–S

––strip–debug

Description

With this option you specify not to include symbolic debug information in the resulting output file.

Example

```
lpcp –S test.o –otest.elf
lpcp ––strip–debug test.o ––output=test.elf
```

The linker generates the object file `test.elf` without symbolic debug information.

-V (--version)

Command line syntax

-V

Description

Display version information. The linker ignores all other options or input files.

Example

```
lpcp -V
lpcp --version
```

The linker does not link any files but displays the following version information:

```
TASKING PCP VX-toolset object linker       vx.yrz Build 000
Copyright years Altium BV                  Serial# 00000000
```

Related information

-

-v (--verbose)

Command line syntax

-v[**v**]
--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.

Example

```
lpcp test.o -dextmem.lsl -ddefault.lsl -lc -lfp -lrt
-v
```

The linker links the file `test.o` and displays the steps it performs.

```
lpcp I437: reading file "extmem.lsl"
lpcp I437: reading file "default.lsl"
lpcp I400: activating link phase
lpcp I403: resolving symbols (task1)
lpcp I404: generating callgraphs (task1)
lpcp I408: executing linker commands (task1)
lpcp I418: finalize linking (task1)
lpcp I401: activating locate phase
...
lpcp I432: finalize locating (task1)
lpcp I402: activating file producing phase
lpcp I433: emitting object files (task1)
lpcp I434: emitting report files (task1)
```

Related information

 -

-w (--no-warnings)

Command line syntax

-w[*nr*[**,***nr*]...]
--no-warnings[**=***nr*[**,***nr*]...]

Description

With this option you can suppresses all warning messages or specific warning messages.

- If you do not specify this option, all warnings are reported.
- If you specify this option but without numbers, all warnings are suppressed.
- If you specify this option with a number, only the specified warnings are suppressed. Separate multiple warnings by commas.

Example

To suppress all warnings:

```
lpcp -w test.o
lpcp --no-warnings test.o
```

To suppress warnings 113 and 114:

```
lpcp -w113,114 test.o
lpcp --no-warnings=113,114 test.o
```

Related information

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

--warnings-as-errors

Command line syntax

--warnings-as-errors[**=***number*,...]

Description

When the linker detects an error or warning, it tries to continue the link process and reports other errors and warnings. When you use this option without arguments, you tell the linker to treat all warnings as errors. This means that the exit status of the linker will be non-zero after the detection of one or more linker warnings. As a consequence, the linker will not produce any output files.

You can also limit this option to specific warnings by specifying a comma-separated list of warning numbers.

Example

```
lpcp --warnings-as-errors test.o
```

When a warning occurs, the linker considers it as an error.

Related information

Linker option **-w** (Suppress some or all warnings)

5.4 CONTROL PROGRAM OPTIONS

The control program **ccpcp** facilitates the invocation of the various components of the PCP toolchain from a single command line.

Some options are interpreted by the control program itself; other options are passed to those programs in the toolchain that accept the option.

Recognized input files

The control program recognizes the following input files:

- Files with a `.c` suffix are interpreted as C source programs and are passed to the compiler.
- Files with a `.asm` suffix are interpreted as hand-written assembly source files which have to be passed to the assembler.
- Files with a `.src` suffix are interpreted as compiled assembly source files. They are directly passed to the assembler.
- Files with a `.a` or `.elb` suffix are interpreted as library files and are passed to the linker.
- Files with a `.o` suffix are interpreted as object files and are passed to the linker.
- Files with a `.out` suffix are interpreted as linked object files and are passed to the locating phase of the linker. The linker accepts only one `.out` file in the invocation.
- An argument with a `.lsl` suffix is interpreted as a linker script file and is passed to the linker.

Normally, the control program tries to compile, assemble, link and locate all source files to absolute object files. There are however, options to suppress the assembler, link or locate stage.

–? (––help)

Command line syntax

–?[**options**]

––help[**=options**]

Description

Displays an overview of all command line options. When you specify the suboption **options**, you receive extended information.

Example

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

```
ccpcp –?
ccpcp
```

Related information


–

--address-size

Command line syntax

--address-size=*addr_size*

Description

If you specify IHEX or SREC with the control option **--format**, you can additionally specify the record length and the address space to be emitted in the output files.

With this option you can specify the size of addresses in bytes (record length). For Intel Hex you can use the values: **1**, **2**, and **4** (default). For Motorola S-records you can specify: **2** (S1 records), **3** (S2 records, default) or **4** bytes (S3 records).

If you do not specify *addr_size*, the default address size is generated.

Example

To create the SREC file `test.s` with S1 records, type:

```
ccpcp --format=SREC --address-size=2
```

Related information

Control program option **--format** (Set linker output format)
Control program option **--space** (Set linker output space name)

Linker option **-o** (Specify an output object file)

-C (--cpu)

Command line syntax

-C*cpu*

Description

With this option you define the target processor for which you create your application. Make sure you choose a target processor *with* pcp!

Based on the specified target processor the control program always includes the correct register files, unless you specify control program option **--no-tasking-sfr**.

Example

To generate the file `test.elf` for the `TC11IB` processor:

```
ccpcp -Ctc1920b test.c
ccpcp --cpu=ttc1920b test.c
```

Related information

Compiler option **-C** (Use SFR definitions for CPU)

Assembler option **-C** (Select CPU)

Section 4.4, *Calling the Compiler*, in Chapter *Using the Compiler* of the *User's Manual*.

--case-insensitive

Command line syntax

--case-insensitive

Description

With this option you tell the control progam not to distinguish between upper and lower case characters. By default upper and lower case characters are considered as different characters. Note that in assembly instruction mnemonics, register names, directives and controls are always treated case insensitive.

Assembly source files that are generated by the compiler must always be assembled and linked case sensitive. When you are writing your own assembly code, you may want to specify the case insensitive mode.

Example

To create the file `test.elf` with case insensitive assembling and linking:

```
ccpcp -c test.c
ccpcp --case-insensitive test.c
```

The assembler and linker now consider upper and lower case characters as being the same. So, for example, the label `LabelName` is the same label as `labelname`.

Related information

Assembler option **--case-sensitive** (Assemble case insensitive)
Linker option **--case-sensitive** (Link case insensitive)

–cs/–co/–cl

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 Stop after C files are compiled to assembly (`.src`)
–co Stop after the files are assembled to object files (`.obj`)
–cl Stop after the files are linked to a linker object file (`.eln`)

To generate the object file `test.o`:

```
ccpcp -c test.c
ccpcp --create=object test.c
```

The control program stops after the file is assembled. It does not link nor locate the generated output.

Related information


–

--check

Command line syntax

--check

Description

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

The compiler/assembler reports any warnings and/or errors.

Example

To check for syntax errors, without generating code:

```
ccpcp --check test.c
```

Related information

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

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

-D (--define)

Command line syntax

-D*macro_name*[**=***macro_definition*]

--define=*macro_name*[**=***macro_definition*]

Description

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

You can specify as many macros as you like. On the command line, use the option **-D** multiple times. If the command line exceeds the length limit of the operating system, you can define the macros in an option file which you then must specify to the control program with the option **-f** *file*.

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

The control program passes the option **-D** (**--define**) to the compiler and the assembler.

Example

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

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

You can now use a macro definition to set the DEMO flag. With the control program this looks as follows:

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

```
ccpcp --define=DEMO test.c
ccpcp --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.

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

Related information

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

-d (--lsl-file)

Command line syntax

-d*file*

--lsl-file=*file*

Description

A linker script file contains vital information about the core for the locating phase of the linker. A linker script file is coded in LSL and contains the following types of information:

- the architecture and derivative definition describe the core's hardware architecture and its internal memory.
- the board specification describes the physical memory available in the system.
- the section layout definition describes how to locate sections in memory.

With this option you specify a linker script file via the control program to the linker. If you do not specify this option, the linker does not use a script file. You can specify the existing file `tc`*`target`*`.lsl` or the name of a manually written linker script file. You can use this option multiple times. The linker processes the LSL files in the order in which they appear on the command line.

Example

To read linker script file information from file `mylslfile.lsl`:

```
ccpcp -dmylslfile.lsl test.obj
ccpcp --lsl-file=mylslfile.lsl test.obj
```

Related information

Section 6.7, *Controlling the Linker with a Script*, in the *User's Manual*.

--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:

```
ccpcp --diag=103
```

This results in message 103 with explanation.

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

```
ccpcp --diag=html:all > ccerrors.html
```

Related information

-

–E (––preprocess)

Command line syntax

–E[*flags*]

––preprocess=[*flags*]

You can set the following flags:

c/**C**	(**+**/**–comments**)	Keep comments
p/**P**	(**+**/**–noline**)	Strip #line source position info

Description

With this option you tell the control program to preprocess the C source.

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

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

With **–Ep** you tell the preprocessor to strip the #line source position information (lines starting with `#line`). These lines are normally processed by the assembler and not needed in the preprocessed output. When you leave these lines out, the output is more orderly to read.

Example

```
ccpcp -EcP test.c -o test.pre
ccpcp --preprocess +comments,-noline test.c
      --output=test.pre
```

The compiler preprocesses the file `test.c` and sends the output to the file `test.pre`. Comments are included but the line source position information is not stripped from the output file.

Related information


-

--error-file

Command line syntax

--error-file

Description

With this option the control program tells the compiler, assembler and linker to redirect error messages to a file. The error file will be named after the input file with extension `.err` (compiler errors), `.ers` (assembler errors) or `.elk` (linker errors).

Example

To write errors to error files instead of stderr, enter:

```
ccpcp --error-file -t test.c
```

Related information

Control program option **--warnings-as-errors** (Warnings as errors)

–F (––no-double)

Command line syntax

–F
––no-double

Description

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

Example

```
ccpcp -F test.c
ccpcp --no-double test.c
```

The file `test.c` is processed where variables of the type double are treated as float in the compilation phase.

Related information

Control program option **––use-double-precision-fp** (Do not replace doubles with floats)

-f (--option-file)

Command line syntax

-f *file*

--option-file=*file*

Description

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

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

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

Format of an option file

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

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

'This has a double quote " embedded'

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

  Note that adjacent strings are concatenated.

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

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

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

Example

Suppose the file `myoptions` contains the following lines:

```
-g
-k
test.c
```

Specify the option file to the control program:

```
ccpcp -f myoptions
cpcp --option-file=myoptions
```

This is equivalent to the following command line:

```
ccpcp -g -k test.c
```

Related information

-

--format

Command line syntax

--format=*format*

You can specify the following formats:

ELF	ELF/DWARF
IHEX	Intel Hex
SREC	Motorola S-records

Description

With this option you specify the output format for the resulting (absolute) object file. The default output format is ELF/DWARF, which can directly be used by the CrossView Pro debugger.

If you choose IHEX or SREC, you can additionally specify the address size of the chosen format (option **--address-size**) and the address space to be emitted (option **--space**).

Example

To generate an Intel Hex output file:

```
ccpcp --format=IHEX test1.c test2.c --output=test.hex
```

Related information

Control program option **--address-size** (For linker IHEX./SREC files)
Control program option **--space** (Set linker output space name)

Linker option **-o** (output file)
Linker option **-C** (generate hex file)

Section 7.1, *ELF/DWARF Object Format*, in Chapter *Object File Formats*.

--fp-trap

Command line syntax

--fp-trap

Description

Default the control program uses the non-trapping floating-point library (`libfp.a`). With this option you tell the control program to use the trapping floating-point library (`libfpt.a`).

If you use the trapping floating-point library, exceptional floating-point cases are intercepted and can be handled separately by an application defined exception handler. Using this library decreases the execution speed of your application.

Example

```
ccpcp --fp-trap test.c
```

Link the trapping floating-point library when generating the object file `test.elf`.

Related information

-

-g (--debug-info)

Command line syntax

-g

--debug-info

Description

With this option you tell the control program to include debug information in the generated object file.

Example

```
ccpcp -g test.c
ccpcp --debug-info test.c
```

The control program includes symbolic debug information in the generated object file `test.elf`.

Related information

-

–I (––include-directory)

Command line syntax

–I*path*

––include–directory=*path*

Description

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

Example

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

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

You can call the control program as follows:

```
ccpcp -Imyinclude test.c
ccpcp --include-directory=myinclude
```

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

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

Related information

Compiler option **–I** (Add directory to include file search path)
Compiler option **–H** (Include file at the start of a compilation)

--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.

Independent 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:

```
ccpcp --iso=90 test.c
```

Related information

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

-k (--keep-output-files)

Command line syntax

-k

--keep-output-files

Description

If an error occurs during the compilation, assembling or linking process, the resulting output file may be incomplete or incorrect. With this option you keep the generated output files when an error occurs.

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

Use this option when you still want to use the generated files. For example when you know that a particular error does not result in a corrupt file, or when you want to inspect the output file, or send it to Altium support.

Example

```
ccpcp -k test.c
ccpcp --keep-output-files test.c
```

When an error occurs during compiling, assembling or linking, the erroneous generated output files will not be removed.

Related information

-

-L (--library-directory / --ignore-default-library-path)

Command line syntax

-L*path*
--library-directory=*path*

-L
--ignore-default-library-path

Description

With this option you can specify the path(s) where your system libraries, specified with the **-l** option, are located. If you want to specify multiple paths, use the option **-L** for each separate path.

The default path is `$(PRODDIR)\lib\pcp1`.

If you specify only **-L** (without a pathname) or the long option **--ignore-default-library-path**, the linker will not search the default path and also not in the paths specified in the environment variables LIBTC1V1_2, LIBTC1V1_3 or LIBTC2. 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 variables LIBTC1V1_2, LIBTC1V1_3 or LIBTC2 when the product was installed.
3. The default directory `$(PRODDIR)\lib\pcp1`, `$(PRODDIR)\lib\pcp15` or `$(PRODDIR)\lib\pcp2`.

Example

Suppose you call the control program as follows:

```
ccpcp test.c -Lc:\mylibs -lcs
ccpcp test.c --library-directory=c:\mylibs -lcs
```

First the linker looks in the directory `c:\mylibs` for library `libc.a` (this option).

If it does not find the requested libraries, it looks in the directory that is set with the environment variables LIBTC1V1_2, LIBTC1V1_3 or LIBTC2.

Then the linker looks in the default directory `$(PRODDIR)\lib\tc1` or `$(PRODDIR)\lib\tc2` for libraries.

Related information

Linker option **–l** (Search also in system library lib*name*)

–l (––library)

Command line syntax

–l*name*

––library=*name*

Description

With this option you tell the linker via the control program to search also in system library `lib`*`name`*`.a`, where *name* is a string. The linker first searches for system libraries in any directories specified with **–L***path*, then in the directories specified with the environment variables LIBTC1V1_2, LIBTC1V1_3 or LIBTC2, unless you used the option **–L** without a directory.

Example

To search in the system library `libfp.a` (floating-point library):

```
ccpcp test.obj mylib.a -lfp
ccpcp test.obj mylib.a --library=fp
```

The linker links the file `test.obj` and first looks in `mylib.a` (in the current directory only), then in the system library `libfp.a` to resolve unresolved symbols.

Related information

Control program option **–L** (Add library directory)

Section 6.4, *Linking with Libraries*, in the *User's Manual*.

–-list-files

Command line syntax

–l*name*

––list–files[**=***file*]

Description

With this option you tell the assembler via the control program to generate a list file for each specified input file. A list file shows the generated object code and the relative addresses. Note that the assembler generates a relocatable object file with relative addresses.

With the optional file you can specify a name for the list file. This is only possible if you specify only one input file to the control program. If you do not specify a file name, or you specify more than one input file, the control program names the generated list file(s) after the specified input file(s) with extension `.lst`.

Note that object files and library files are not counted as input files.

Related information

Assembler option **––list–file** (Generate list file)
Assembler option **––list–format** (Format list file)

-n (--dry-run)

Command line syntax

-n
--dry-run

Description

With this option you put the control program *verbose* mode. The control program prints the invocations of the tools it would use to process the files.

Example

To see how the control program will invoke the tools it needs to process the file `test.c`:

```
ccpcp -n test.c
ccpcp --dry-run test.c
```

The control program only displays the invocations of the tools it would use to create the final object file but does not actually perform the steps.

Related information

Control program option **-v** (Verbose output)

--no-default-libraries

Command line syntax

--no-default-libraries

Description

Default the control program specifies the standard C libraries and run-time library to the linker.

With this option you tell the control program not to specify the standard C libraries and run-time library to the linker.

In this case you must specify the libraries you want to link to the linker with the option **–l***library_name*. The control program recognizes the option **–l** as an option for the linker.

Example

```
ccpcp --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:

```
ccpcp --no-default-libraries -lmy test.c
```

Related information

Linker option **–l** (Search also in system library `libx.a`)

--no-map-file

Command line syntax

--no-map-file

Description

By default the control program generates a linker map file (`.map`).

A linker map file is a text file that shows how the linker has mapped the sections and symbols from the various object files (`.obj`) to the linked object file. A locate part shows the absolute position of each section. External symbols are listed per space with their absolute address, both sorted on symbol and sorted on address.

With this option you prevent the generation of a map file.

Example

To prevent the generation of the linker map file `test.map`:

```
ccpcp --no-map-file test.c
```

Related information

Linker option **-M** (Generate map file)

--no-tasking-sfr

Command line syntax

--no-tasking-sfr

Description

Normally, the compiler and assembler include a special function register (SFR) file before compiling. This file is automatically selected based on the target you select on the **Processor definition** page of the Processor options (compiler option **-C**).

With this option the compiler and assembler do *not* automatically include a register file.

Use this option if you want to use your own set of SFR files.

Example

```
ccpcp -Ctc11ib --no-tasking-sfr test.c
```

The register file `regtc11ib.sfr` is not included.

Related information

Compiler option **-C** (Use SFR definitions for CPU)

-o (--output)

Command line syntax

-o*file*

--output=*file*

Description

By default, the control program generates a file with the same basename as the first specified input file. With this option you specify another name for the resulting absolute object file.

Example

```
ccpcp test.c prog.c
```

The control program generates an ELF/DWARF object file (default) with the name `test.elf`.

To generate the file `result.elf`:

```
ccpcp -oresult.elf test.c prog.c
ccpcp --output=result.elf test.c prog.c
```

Related information

-

--space

Command line syntax

--space=*space_name*

Description

If you specify IHEX or SREC with the control option **--format**, you can additionally specify the record length and the address space to be emitted in the output files.

With this option you can specify which address space must be emitted. With the argument *space_name* you can specify the name of the address space. The name of the output file will be *filename* with the extension `.hex` or `.s`.

If you do not specify *space_name*, the default address space is emitted. In this case the name of the output file will be *filename_spacename* with the extension `.hex` or `.s`.

Example

To create the IHEX file `test.hex`, type:

```
ccpcp --format=IHEX --space=far test.c
```

If the specified memory space does not exist, the control program emits the default space name and reflects this in the output file name.

Related information

Control program option **--format** (Set linker output format)

Linker option **-o** (Specify an output object file)

--static

Command line syntax

--static

Description

This option is directly passed to the compiler.

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

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

Example

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

Related information

-

-t (--keep-temporary-files)

Command line syntax

-t

--keep-temporary-files

Description

By default, the control program removes intermediate files like the `.src` file (result of the compiler phase) and the `.eln` file (result of the linking phase).

With this option you tell the control program to keep temporary files it generates during the creation of the absolute object file.

Example

To keep all temporary files:

```
ccpcp -t test.c
ccpcp --keep-temporary-files test.c
```

The control program keeps all intermediate files it generates while creating the absolute object file `test.elf`.

Related information

-

-U (--undefine)

Command line syntax

-U*macro_name*
--undefine=*macro_name*

Description

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

This option is for example useful to undefine predefined macros.

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

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

The control program passes the option **-U** (**--undefine**) to the compiler.

Example

To undefine the predefined macro __TASKING__:

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

Related information

Control Pogram option **-D** (Define preprocessor macro)

-V (--version)

Command line syntax

-V
--version

Description

Display version information. The control program ignores all other options or input files.

Example

```
ccpcp -V
ccpcp --version
```

The control program does not call any tools but displays the following version information:

```
TASKING PCP VX-toolset control program       vx.yrz Build nnn
Copyright 2003-year Altium BV                Serial# 00000000
```

Related information


-

-v (--verbose)

Command line syntax

-v
--verbose

Description

With this option you put the control program in *verbose* mode. With the option **-v** the control program performs it tasks while it prints the steps it performs to `stdout`.

Example

```
ccpcp -v test.c
ccpcp --verbose test.c
```

The control program processes the file `test.c` and displays the invocations of the tools it uses to create the final object file

Related information

Control program option **-n** (Verbose output and suppress execution)

-W*tool* (--pass)

Command line syntax

-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 the option itself, but specifies it directly to the tool which the control program calls.

Example

```
ccpcp -Wl-r test.c
```

The control program does not use the option **-r** but calls the linker with the option **-r** (`lpcp -r`).

Related information

-

-w (--no-warnings)

Command line syntax

-w[*nr*]
--no-warnings[**=***nr*]

Description

With this option suppresses all warning messages or a specific warning. If you do not specify this option, all warnings are reported.

Example

To suppress all warnings:

```
ccpcp -w test.c
ccpcp --no-warnings test.c
```

To suppress warnings 100:

```
ccpcp -w100 test.c
ccpcp --no-warnings=100 test.c
```

Related information

Control program option **--warnings-as-errors** (Warnings as errors)

--warnings-as-errors

Command line syntax

--warnings-as-errors

Description

With this option you tell the control program to treat warnings as errors.

Example

```
ccpcp --warnings-as-errors test.c
```

When a warning occurs, the control program considers it as an error.

Related information

Control program option **-w** (Suppress all warnings)

5.5 MAKE UTILITY OPTIONS

You can use the make utility **mkpcp** from the command line to build your project.

The invocation syntax is:

mkpcp [*option...*] [*target...*] [*macro=def*]

This section describes all options for the make utility.

Defining Macros

Command line syntax

macro=*definition*

Description

With this argument you can define a macro and specify it to the make utility.

A macro definition remains in existence during the execution of the makefile, even when the makefile recursively calls the make utility again. In the recursive call, the macro acts as an environment variable. This means that it is overruled by definitions in the recursive call. Use the option **-e** to prevent this.

You can specify as many macros as you like. If the command line exceeds the limit of the operating system, you can define the macros in an *option file* which you then must specify to the compiler with the option **-m** *file*.

Defining macros on the command line is, for example, useful in combination with conditional processing as shown in the example below.

Example

Consider the following makefile with conditional rules to build a demo program and a real program:

```
ifdef DEMO     # the value of DEMO is of no importance
  real.out : demo.o
             lpcp demo.o main.o -lc -lfp -lrt
else
  real.out : real.o
             lpcp real.o main.o -lc -lfp -lrt
endif

real.elf   : real.out
             lpcp -FELF -oreal.elf real.out
```

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

```
mkpcp real.elf DEMO=1
```

In both cases the absolute obect file `real.elf` is created but depending on the DEMO flag it is linked with `demo.o` or with `real.o`.

Related information

Make utility option **–e** (Environment variables override macro definitions)
Make utility option **–m** (Name of invocation file)

–?

Command line syntax

```
–?
```

Description

Displays an overview of all command line options.

Example

The following invocation displays a list of the available command line options:

```
mkpcp –?
```

Related information


–

-a

Command line syntax

-a

Description

Normally the make utility rebuilds only those files that are out of date. With this option you tell the make utility to rebuild *all* files, without checking whether they are out of date.

Example

```
mkpcp -a
```

Rebuilds all your files, regardless of whether they are out of date or not.

Related information

-

–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 **mkpcp**.

With the option **–DD** not only the lines of the makefile are printed but also the lines of the `mkpcp.mk` file (implicit rules).

Example

```
mkpcp -D
```

Each line of the makefile that is read by the make utility is printed to standard output (usually your screen).

Related information


-

-d/-dd

Command line syntax

-d
-dd

Description

With the option **-d** the make utility shows which files are out of date and thus need to be rebuild. The option **-dd** gives more detail than the option **-d**.

Example

```
mkpcp -d
```

Shows which files are out of date and rebuilds them.

Related information

-

-e

Command line syntax

-e

Description

If you use macro definitions, they may overrule the settings of the environment variables.

With the option **-e**, the settings of the environment variables are used even if macros define otherwise.

Example

```
mkpcp -e
```

The make utility uses the settings of the environment variables regardless of macro definitions.

Related information

-

-err

Command line syntax

-err *file*

Description

With this option the make utility redirects error messages and verbose messages to a specified file.

With the option **-s** the make utility only displays error messages.

Example

```
mkpcp -err error.txt
```

The make utility writes messages to the file `error.txt`.

Related information

Make utility option **-s** (Do not print commands before execution)

–f

Command line syntax

–f *my_makefile*

Description

Default the make utility uses the file `makefile` to build your files.

With this option you tell the make utility to use the specified file instead of the file `makefile`. Multiple **–f** options act as if all the makefiles were concatenated in a left-to-right order.

Example

```
mkpcp mymake
```

The make utility uses the file `mymake` to build your files.

Related information

–

-G

Command line syntax

-G *path*

Description

Normally you must call the make utility **mkpcp** 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 `\currdir\myfiles`. When your current directory is `\currdir`, you can call the make utility as follows:

```
mkpcp -G myfiles
```

Related information

-

–i

Command line syntax

–i

Description

When an error occurs during the make process, the make utility exits with a certain exit code.

With the option **–i**, the make utility exits without an error code, even when errors occurred.

Example

```
mkpcp -i
```

The make utility exits without an error code, even when an error occurs.

Related information

-

–K

Command line syntax

–K

Description

With this option the make utility keeps temporary files it creates during the make process. The make utility stores temporary files in the directory that you have specified with the environment variable TMPDIR or in the default 'temp' directory of your system when the TMPDIR variable is not specified.

Example

```
mkpcp -K
```

The make utility preserves all temporary files.

Related information

Section 1.3.1, *Configuring the Command Line Environment*, in Chapter *Software Installation* of the *User's Manual*.

–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

```
mkpcp -k
```

If the make utility encounters an error, it stops building the current target but proceeds with the other targets that are defined in the makefile.

Related information

Make utility option **–S** (Undo the effect of **–k**)

-m

Command line syntax

-m *file*

Description

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

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

You can specify the option **-m** multiple times.

Format of an option file

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

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

'This has a double quote " embedded'

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

  Note that adjacent strings are concatenated.

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

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

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

Example

Suppose the file `myoptions` contains the following lines:

```
-k
-err errors.txt
test.elf
```

Specify the option file to the make utility:

```
mkpcp -m myoptions
```

This is equivalent to the following command line:

```
mkpcp -k -err errors.txt test.elf
```

Related information

-

-n

Command line syntax

-n

Description

With this option you tell the make utility to perform a *dry run*. The make utility shows what it would do but does not actually perform these tasks.

This option is for example useful to quickly inspect what would happen if you call the make utility.

Example

```
mkpcp -n
```

The make utility does not perform any tasks but displays what it would do if called without the option **-n**.

Related information

Make utility option **-s** (Do not print commands before execution)

–p

Command line syntax

–p

Description

Normally, if a command in a target rule in a makefile returns an error or when the target construction is interrupted, the make utility removes that target file. With this option you tell the make utility to make all target files precious. This means that all dependency files are never removed.

Example

```
mkpcp -p
```

The make utility never removes target dependency files.

Related information

Special target .PRECIOUS in section 7.3.2, *Writing a Makefile* in Chapter *Using the Utilities* of the *Reference Manual*.

-q

Command line syntax

-q

Description

With this option the make utility does not perform any tasks but only returns an error code. A zero status indicates that all target files are up to date, a non-zero status indicates that some or all target files are out of date.

Example

```
mkpcp -q
```

The make utility only returns an error code that indicates whether all target files are up to date or not. It does not rebuild any files.

Related information

-

-r

Command line syntax

-r

Description

When you call the make utility, it first reads the implicit rules from the file **mkpcp.mk**, then it reads the makefile with the rules to build your files. (The file **mkpcp.mk** is located in the **\etc** directory of the PCP toolchain.)

With this option you tell the make utility *not* to read **mkpcp.mk** and to rely fully on the make rules in the makefile.

Example

```
mkpcp -r
```

The make utility does not read the implicit make rules in **mkpcp.mk**.

Related information

-

-S

Command line syntax

-S

Description

With this option you cancel the effect of the option **-k**. This is never necessary except in a recursive make where the option **-k** might be inherited from the top-level make via MAKEFLAGS or if you set the option **-k** in the environment variable MAKEFLAGS.

Example

```
mkpcp -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 **mkpcp** in the makefile.

Related information

Make utility option **-k** (On error, abandon the work for the current target only)

–s

Command line syntax

–s

Description

With this option you tell the make utility to perform its tasks without printing the commands it executes. Error messages are normally printed.

Example

```
mkpcp -s
```

The make utility rebuilds your files but does not print the commands it executes during the make process.

Related information

Make utility option **–n** (Perform a dry run)

-t

Command line syntax

-t

Description

With this option you tell the make utility to *touch* the target files, bringing them up to date, rather than performing the rules to rebuild them.

Example

```
mkpcp -t
```

The make utility updates out-of-date files by giving them a new date and time stamp. The files are not actually rebuild.

Related information

-

–time

Command line syntax

–time

Description

With this option you tell the make utility to display the current date and time on standard output.

Example

```
mkpcp –time
```

The make utility displays the current date and time and updates out-of-date files.

Related information

-

-V

Command line syntax

-V

Description

Display version information. The make utility ignores all other options or input files.

Example

```
mkpcp -v
```

The make utility does not perform any tasks but displays the following version information:

```
TASKING PCP VX-toolset program builder       vxx.yrz Build nnn
Copyright year Altium BV                     Serial# 00000000
```

Related information

-

-W

Command line syntax

-W *target*

Description

With this option the make utility considers the specified target file always as up to date and will not rebuild it.

Example

```
mkpcp -W test.elf
```

The make utility rebuilds out of date targets in the makefile except the file `test.elf` which is considered now as up to date.

Related information

-

-w

Command line syntax

-w

Description

With this option the make utility sends error messages and verbose messages to standard out. Without this option, the make utility sends these messages to standard error.

This option is only useful on UNIX systems.

Example

```
mkpcp -w
```

The make utility sends messages to standard out instead of standard error.

Related information

-

5.6 ARCHIVER OPTIONS

The archiver and library maintainer **arpcp** is a tool to build library files and it offers the possibility to replace, extract and remove modules from an existing library.

The invocation syntax is:

arpcp *key_option* [*sub_option...*] *library* [*object_file*]

This section describes all options for the archiver. Some suboptions can only be used in combination with certain key options. They are described together. Suboptions that can always be used are described separately.

The archiver is a command line tool so there are no equivalent options in EDE.

Description	Option	Suboption
Display an overview of all options	**–?**	
Display version information	**–V**	
Read options from *file*	**–f** *file*	
Print object module to standard output	**–p**	
Suppress warnings above level *n*	**–w***n*	
Main functions		
Delete object module from library	**–d**	**–v**
Move object module to another position	**–m**	**–a –b –v**
Replace or add an object module	**–r**	**–a –b –c –u –v**
Print a table of contents of the library	**–t**	**–s0 –s1**
Extract an object module from the library	**–x**	**–v**

Table 5-1: Overview of archiver options and suboptions

-?

Command line syntax

```
-?
```

Description

Displays an overview of all command line options.

Example

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

```
arpcp -?
arpcp
```

Related information

-

-d

Command line syntax

–d [**–v**]

Description

Delete the specified object modules from a library. With the suboption **–v** the archiver shows which files are removed.

–v Verbose: the archiver shows which files are removed.

Example

```
arpcp –d lib.a obj1.o obj2.o
```

The archiver deletes `obj1.o` and `obj2.o` from the library `lib.a`.

```
arpcp –d –v lib.a obj1.o obj2.o
```

The archiver deletes `obj1.o` and `obj2.o` from the library `lib.a` and displays which files are removed.

Related information

 –

-f

Command line syntax

-f *file*

Description

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

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.

If you use '-' instead of a filename it means that the options are read from stdin.

Format of an option file

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

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

'This has a double quote " embedded'

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

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

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

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

Example

Suppose the file `myoptions` contains the following lines:

```
-x mylib. obj1.o
-w5
```

Specify the option file to the archiver:

```
arpcp -f myoptions
```

This is equivalent to the following command line:

```
arpcp -x mylib. obj1.o -w5
```

Related information

-

-m

Command line syntax

-m [**-a** *posname*] [**-b** *posname*]

Description

Move the specified object modules to another position in the library.

The ordering of members in a library can make a difference in how programs are linked if a symbol is defined in more than one member.

Default, the specified members are moved to the end of the archive. Use the suboptions **-a** or **-b** to move them to a specified place instead.

-a *posname* Move the specified object module(s) after the existing module *posname*.

-b *posname* Move the specified object module(s) before the existing module *posname*.

Example

Suppose the library `lib.a` contains the following objects (see option **-t**):

```
obj1.o
obj2.o
obj3.o
```

To move `obj1.o` to the end of `lib.a`:

```
arpcp -m lib.a obj1.o
```

To move `obj3.o` just before `obj2.o`:

```
arpcp -m -b obj3.o lib.a obj2.o
```

The library `lib.a` after these two invocations now looks like:

```
obj3.o
obj2.o
obj1.o
```

Related information

Archiver option **-t** (Print library contents)

-p

Command line syntax

-p

Description

Print the specified object module(s) in the library to standard output.

This option is only useful when you redirect or pipe the output to other files or tools that serve your own purposes. Normally you do not need this option.

Example

```
arpcp -p lib.a obj1.o > file.o
```

The archiver prints the file `obj1.o` to standard output where it is redirected to the file `file.o`. The effect of this example is very similar to extracting a file from the library but in this case the 'extracted' file gets another name.

Related information

-

–r

Command line syntax

–r [**–a** *posname*] [**–b** *posname*] [**–c**] [**–u**] [**–v**]

Description

You can use the option **–r** for several purposes:

- Adding new objects to the library
- Replacing objects in the library with the same object of a newer date
- Creating a new library

The option **–r** normally *adds* a new module to the library. However, if the library already contains a module with the specified name, the existing module is *replaced*. If you specify a library that does not exist, the archiver *creates* a new library with the specified name.

If you add a module to the library without specifying the suboption **–a** or **–b**, the specified module is added at the end of the archive. Use the suboptions **–a** or **–b** to insert them to a specified place instead.

–a *posname*	Add the specified object module(s) after the existing module *posname*.
–b *posname*	Add the specified object module(s) before the existing module *posname*.
–c	Create a new library without checking whether it already exists. If the library already exists, it is overwritten.
–u	Insert the specified object module only if it is newer than the module in the library.
–v	Verbose: the archiver shows which files are removed.

The suboptions **–a** or **–b** have no effect when an object is added to the library.

Examples

Suppose the library lib.a contains the following objects (see option **–t**):

```
obj1.o
```

To add `obj2.o` to the end of `lib.a`:

```
arpcp –r lib.a obj2.o
```

To insert `obj3.o` just before `obj2.o`:

```
arpcp –r –b obj2.o lib.a obj3.o
```

The library `lib.a` after these two invocations now looks like:

```
obj1.o
obj3.o
obj2.o
```

Creating a new library

To *create a new library file*, add an object file and specify a library that does not yet exist:

```
arpcp –r obj1.o newlib.a
```

The archiver creates the library `newlib.a` and adds the object `obj1.o` to it.

To *create a new library file and overwrite an existing library*, add an object file and specify an existing library with the supoption **–c**:

```
arpcp –r –c obj1.o lib.a
```

The archiver overwrites the library `lib.a` and adds the object `obj1.o` to it. The new library `lib.a` only contains `obj1.o`.

Related information

Archiver option **–t** (Print library contents)

-t

Command line syntax

-t [**-s0**|**-s1**]

Description

Print a table of contents of the library to standard out. With the suboption **-s** the archiver displays all symbols per object file.

-s0 Displays per object the library in which it resides, the name of the object itself and all symbols in the object.

-s1 Displays only the symbols of all object files in the library.

Example

```
arpcp -t lib.a
```

The archiver prints a list of all object modules in the library `lib.a`.

```
arpcp -t -s0 lib.a
```

The archiver prints per object all symbols in the library. This looks like:

```
prolog.o
   symbols:
lib.a:prolog.o:___Qabi_callee_save
lib.a:prolog.o:___Qabi_callee_restore
div16.o
   symbols:
lib.a:div16.o:___udiv16
lib.a:div16.o:___div16
lib.a:div16.o:___urem16
lib.a:div16.o:___rem16
```

Related information

-

-V

Command line syntax

-V

Description

Display version information. The archiver ignores all other options or input files.

Example

```
arpcp -V
```

The archiver does not perform any tasks but displays the following version information:

```
TASKING PCP VX-toolset ELF archiver      vxx.yrz Build nnn
Copyright year Altium BV                 Serial# 00000000
```

Related information

 -

-x

Command line syntax

-x [**-o**] [**-v**]

Description

Extract an existing module from the library.

-o Give the extracted object module the same date as the last-modified date that was recorded in the library.

Without this suboption it receives the last-modified date of the moment it is extracted.

-v Verbose: the archiver shows which files are extracted.

Example

To extract the file `obj.o` from the library `lib.a`:

```
arpcp -x lib.a obj1.o
```

If you do not specify an object module, all object modules are extracted:

```
arpcp -x lib.a
```

Related information

-

–w

Command line syntax

–w*level*

Description

With this suboption you tell the archiver to suppress all warnings above the specified level. The level is a number between 0 - 9.

The level of a message is printed between parentheses after the warning number. If you do not use the **–w** option, the default warning level is 8.

Example

To suppresses warnings above level 5:

```
arpcp -x -w5 lib.a obj1.o
```

Related information

-

CHAPTER 6

LIST FILE FORMATS

TASKING

9 CHAPTER

6.1 ASSEMBLER LIST FILE FORMAT

The assembler list file is an additional output file of the assembler that contains information about the generated code.

The list file consists of a page header and a source listing.

Page header

The page header consists of four lines:

```
TASKING PCP VX-toolset Assembler vx.yrz Build nnn SN 00000000
This is the page header title                              Page 1

ADDR CODE      CYCLES  LINE SOURCE LINE
```

The first line contains information about the assembler name, version number and serial number. The second line contains a title specified by the TITLE (first page) assembler directive and a page number. The third line is empty. The fourth line contains the heading of the source listing.

Source listing

The following is a sample part of a listing. An explanation of the different columns follows below.

```
ADDR CODE      CYCLES  LINE SOURCE LINE
                         .
                         .
0000 93C0rrrr            26         ldl.il  r7,@DPTR(_PCP__99900...
0002 54rr                27         st.pi   r2,[_PCP__999001__1]
0003 9340rrrr            28         ldl.il  r5,@LO(_PCP__lc_ub__...
0005 93C0rrrr            29         ldl.il  r7,@DPTR(_PCP__data_...
0007 55rr                30         st.pi   r5,[_PCP__data__printf]
0008 9240rrrr            31         ldl.il  r1,@LO(_PCP__1_str)
000A 93C0rrrr            32         ldl.il  r7,@DPTR(_PCP__99900...
000C 52rr                33         ld.pi   r2,[_PCP__999001__1]
000D E800rrrr            34         jg      _PCP_printf
                         .
                         .
0000 RESERVED            86         .space  1
```

The meaning of the various 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".
`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 5.7, *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.

6.2 LINKER MAP FILE FORMAT

The linker map file is an additional output file of the linker that shows how the link phase has mapped the sections and symbols from the various object files (`.o`) to output sections. The locate part shows the absolute position of each section. External symbols are listed per space with their absolute address, both sorted on symbol and sorted on address.

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.o | libc.a       | _START                        |
| hello.o  |              |                               |
| printf.o | libc.a       | printf                        |
+-----------------------------------------------------------+


***************************************  Link Part  ***********************************
+-------------------------------------------------------------------------------------+
| [in] File | [in] Section     | [in] Size  | [out] Offset | [out] Section    |
|=====================================================================================|
| hello.o   | .text.hello.main | 0x00000014 | 0x00000000   | .text.hello.main |
|-------------------------------------------------------------------------------------|
| cstart.o  | .text.libc       | 0x00000202 | 0x00000000   | .text.libc       |
| strcpy.o  | .text.libc       | 0x00000024 | 0x00000204   |                  |
| printf.o  | .text.libc       | 0x0000002c | 0x00000800   |                  |
|-------------------------------------------------------------------------------------|
| cstart.o  | .text.libc.reset | 0x00000008 | 0x00000000   | .text.libc.reset |
+-------------------------------------------------------------------------------------+


*********************************  Module Local Symbols Part  ***************************

* Symbol translation (sorted on name)
=====================================

+ Scope "./hello.o"
+-----------------------------------------------+
| Name              | Address    | Space        |
|===============================================|
| hello.src         | 0x00000000 | -            |
| .rodata.hello     | 0xa00000e0 | spe:tc:linear |
| .text.hello.main  | 0xa00000f8 |              |
| .zdata.hello      | 0xd0000000 | spe:tc:abs18 |
| .zrodata.hello    | 0xa0000008 |              |
+-----------------------------------------------+
```

```
********************************** Cross Reference Part  ******************************

+-------------------------------------------------------------------------+
| Definition file | Definition section | Symbol        | Referenced in |
|=========================================================================|
| _doprint_int.o  | .text.libc         | __printf_int2 | printf_int.o  |
| _doprint_int.o  | .text.libc         | _doprint      | printf.o      |
| cstart.o        | .text.libc         | _start        | hello.o       |
| cstart.o        | .text.libc.reset   | _START        |               |
| hello.o         | .text.hello.main   | main          | cstart.o      |
| printf.o        | .text.libc         | printf        | hello.o       |
+-------------------------------------------------------------------------+


********************************** Call Graph Part  ********************************


************************************ Overlay Part  *********************************


************************************ Locate Part  **********************************

* Task entry address
====================
+---------------------------------+
| symbol           | _START       |
| absolute address | 0xa0000000 |
+---------------------------------+

* Section translation
======================

+ Space spe:tc:linear

+-------------------------------------------------------------------------------------+
| Chip      | Group     | Section              | Size (MAU) | Space addr | Chip addr  |
|=====================================================================================|
| ext_c     |           | .text.libc.reset     | 0x00000008 | 0xa0000000 | 0x00000000 |
|           |           | [.data.libc]         | 0x000000cc | 0xa0000010 | 0x00000010 |
|           |           | [.zdata.hello]       | 0x00000004 | 0xa00000dc | 0x000000dc |
|           |           | .rodata.hello        | 0x0000000c | 0xa00000e0 | 0x000000e0 |
|           |           | .rodata.libc         | 0x0000000c | 0xa00000ec | 0x000000ec |
|           |           | .text.hello.main     | 0x00000014 | 0xa00000f8 | 0x000000f8 |
|           |           | .text.libc.csa_areas | 0x00000008 | 0xa000010c | 0x0000010c |
|           |           | table                | 0x00000034 | 0xa0000114 | 0x00000114 |
|           | libraries | .text.libc           | 0x00000eae | 0xa0004000 | 0x00004000 |
+-------------------------------------------------------------------------------------+

+ Space spe:tc:abs18

+-------------------------------------------------------------------------------------+
| Chip      | Group     | Section              | Size (MAU) | Space addr | Chip addr  |
|=====================================================================================|
| ext_c     |           | .zrodata.hello       | 0x00000006 | 0xa0000008 | 0x00000008 |
| spe:dsram |           | .zdata.hello         | 0x00000004 | 0xd0000000 | 0x00000000 |
+-------------------------------------------------------------------------------------+


* Symbol translation (sorted on name)
======================================
+-----------------------------------------------+
| Name            | Address    | Space          |
|===============================================|
| _A8_DATA_       | 0x00000000 | spe:tc:linear  |
| _Exit           | 0xa0004170 |                |
| _START          | 0xa0000000 |                |
| _lc_gb_int_tab  | 0x00000000 |                |
| _lc_ge_int_tab  | 0x00000000 |                |
| main            | 0xa00000f8 |                |
| world           | 0xd0000000 | spe:tc:abs18   |
+-----------------------------------------------+
```

```
* Symbol translation (sorted on address)
=======================================
+-------------------------------------------+
| Address    | Name           | Space       |
|===========================================|
| 0x00000000 | _lc_ge_int_tab | spe:tc:linear |
| 0x00000000 | _lc_gb_int_tab |             |
| 0x00000000 | _A8_DATA_      |             |
| 0xa0000000 | _START         |             |
| 0xa00000f8 | main           |             |
| 0xa0004170 | _Exit          |             |
| 0xd0000000 | world          | spe:tc:abs18  |
+-------------------------------------------+


************************************  Memory Part  *********************************

* Address range usage at space level
====================================
+-------------------------------------------------------------------------------------+
| Name            | Total      | Used        %  | Free        %  | > free gap  %  |
|=====================================================================================|
| spe:tc:abs18    | 0x00008000 | 0x00000146   1 | 0x00007eba  99 | 0x00004000  50 |
| spe:tc:abs24    | 0x50024000 | 0x00022820   1 | 0x500017e0  99 | 0x2ff0df0a  59 |
| spe:tc:csa      | 0x00006000 | 0x00001004  17 | 0x00004ffc  83 | 0x00004fc0  83 |
| spe:tc:linear   | 0x00085000 | 0x00021788  26 | 0x00063878  74 | 0x0004f878  59 |
| spe:tc:pcp_code | 0x00002000 | 0x00000000   0 | 0x00002000 100 | 0x00002000 100 |
| spe:tc:pcp_data | 0x00000400 | 0x00000000   0 | 0x00000400 100 | 0x00000400 100 |
+-------------------------------------------------------------------------------------+

* Address range usage at memory level
====================================
+-------------------------------------------------------------------------------+
| Name        | Total      | Used        %  | Free        %  | > free gap  %  |
|===============================================================================|
| ext_c       | 0x00080000 | 0x00000ff4   1 | 0x0007f00c  99 | 0x0007b152  96 |
| ext_c2      | 0x00080000 | 0x00000000   0 | 0x00080000 100 | 0x00080000 100 |
| ext_d       | 0x00070000 | 0x00020784  29 | 0x0004f87c  71 | 0x0004f878  71 |
| spe:brom    | 0x00008000 | 0x00000000   0 | 0x00008000 100 | 0x00008000 100 |
| spe:csram   | 0x00006000 | 0x00000000   0 | 0x00006000 100 | 0x00006000 100 |
| spe:dsram   | 0x00006000 | 0x00001004  17 | 0x00004ffc  83 | 0x00004fc0  83 |
| spe:fpidram | 0x00004000 | 0x00000000   0 | 0x00004000 100 | 0x00004000 100 |
| spe:pcode   | 0x00004000 | 0x00000000   0 | 0x00004000 100 | 0x00004000 100 |
| spe:pram    | 0x00001000 | 0x00000000   0 | 0x00001000 100 | 0x00001000 100 |
| vecttable   | 0x00002400 | 0x000000a4   2 | 0x0000235c  98 | 0x00002000  88 |
+-------------------------------------------------------------------------------+


*********************************  Linker Script File Part  *****************************


***********************************  Locate Rule Part  ********************************
+-----------------------------------------------------------------------------------------+
| Address space | Type         | Properties | Sections                                    |
|=========================================================================================|
| spe:tc:linear | absolute     | 0xa0004000 | .text.libc                                  |
| spe:tc:abs18  | clustered    |            | .zdata.hello                                |
| spe:tc:linear | clustered    |            | .data.libc                                  |
| spe:tc:linear | ordered      |            | .text.trapvec.000 < .text.trapvec.001 ...   |
| spe:tc:abs18  | unrestricted |            | .zrodata.hello                              |
| spe:tc:linear | unrestricted |            | ustack                                      |
+-----------------------------------------------------------------------------------------+
```

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 (`.o`) to output sections.

`[in] File`	The name of an input object file.
`[in] Section`	A section name from the input object file.
`[in] Size`	The size of the input section.
`[out] Offset`	The offset relative to the start of the output section.
`[out] Section`	The resulting output section name.

Module Local Symbols Part

This part of the map file shows a table for each local scope within an object file. Each table has three columns, 1 the 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.

Call Graph Part

This part of the map file contains a schematic overview that shows how (library) functions call each other. To obtain call graph information, the assembly file must contain `.CALLS` directives which you must manually add to the assembly source.

Overlay Part

This part is empty for the PCP.

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 (`tc*.lsl`). The names are constructed of the `derivative` name followed by a colon ':', the `core` name, another colon ':' and the `space` name. For example: `spe:tc:linear`
`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.

Locator Part: Symbol translation

This part of the map file lists all external symbols per address space name, both sorted on symbol name and sorted on address.

`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 (`tc*.lsl`). The names are constructed of the `derivative` name followed by a colon ':', the `core` name, another colon ':' and the `space` name. For example: `spe:tc:linear`

Memory Part

This part of the map file shows the memory usage in totals and percentages for spaces and chips. The largest free block of memory per space and per chip is also shown.

By default this part is not shown in the map file. You have to turn this part on manually with linker option **–mm** (memory usage info).

Linker Script File Part

This part of the map file shows the processor and memory information of the linker script file.

By default this part is not shown in the map file. You have to turn this part on manually with linker option **–ms** (processor and memory info). You can print this information to a separate file with linker option **--lsl-dump**.

Locate Rule Part

This part of the map file shows the rules the linker uses to locate sections.

`Address space` The names of the address spaces as defined in the linker script file (`*.lsl`). The names are constructed of the `derivative` name followed by a colon ':', the `core` name, another colon ':' and the `space` name. For example: `spe:tc:linear`

`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 address` The section must be located at the address shown in the Properties column

`address range`	The section must be located in the union of the address ranges shown in the Properties column; end addreses are not included in the range.
`address range size`	The sections must be located in some address range with size not larger than shown in the Properties column; the second number in that field is the alignment requirement for the address range.
`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 6.9, *Generating a Map File*, in Chapter *Using the Linker* of the *User's Manual*.

Linker option **-M** (Generate map file)

CHAPTER 7

OBJECT FILE FORMATS

TASKING

1 CHAPTER

7.1 ELF/DWARF OBJECT FORMAT

The PCP toolchain by default produces objects in the ELF/DWARF 2 (`.elf`) format.

The ELF/DWARF 2 Object Format for the PCP toolchain follows the convention as described in the *PCP Embedded Application Binary Interface* [2000, Infineon].

For a complete description of the ELF and DWARF formats, please refer to the *Tool Interface Standard (TIS)*.

7.2 MOTOROLA S-RECORD FORMAT

With the linker option **–o***filename***:SREC** option the linker produces output in Motorola S-record format with three types of S-records: S0, S2 and S8. With the options **–o***filename***:SREC:2** or **–o***filename***:SREC:4** option you can force other types of S-records. They have the following layout:

S0 - record

'S' '0' <*length_byte*> <2 bytes 0> <*comment*> <*checksum_byte*>

A linker generated S-record file starts with a S0 record with the following contents:

length_byte : 0x6
comment : lpcp (PCP linker)
checksum : 0xB6

```
          l p c p
S00700006C70637049
```

The S0 record is a comment record and does not contain relevant information for program execution.

The length_byte represents the number of bytes in the record, not including the record type and length byte.

The checksum is calculated by first adding the binary representation of the bytes following the record type (starting with the length_byte) to just before the checksum. Then the one's complement is calculated of this sum. The least significant byte of the result is the checksum. The sum of all bytes following the record type is 0xFF.

S1 - record

With the linker option **–o***filename***:SREC:2**, the actual program code and data is supplied with S1 records, with the following layout:

'S' '1' <*length_byte*> <*address*> <*code bytes*> <*checksum_byte*>

This record is used for 2-byte addresses.

Example:

```
S1130250F03EF04DF0ACE8A408A2A013EDFCDB00E6
  | |    |                                |_ checksum
  | |    |_ code
  | |_ address
  |_ length
```

The 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**, which is the default, the actual program code and data is supplied with S2 records, with the following layout:

'S' '2' *<length_byte> <address> <code bytes> <checksum_byte>*

For the PCP the linker generates 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**, the actual program code and data is supplied with S3 records, with the following layout:

'S' '3' *<length_byte> <address> <code bytes> <checksum_byte>*

This record is used for 4-byte addresses.

Example:

```
S3070000FFFE6E6825
  | |       |   |_ checksum
  | |       |_ code
  | |_ address
  |_ length
```

The 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**, at the end of an S-record file, the linker generates an S7 record, which contains the program start address. S7 is the corresponding termination record for S3 records.

Layout:

'S' '7' <*length_byte*> <*address*> <*checksum_byte*>

Example:

```
S70500006E6824
  | |       |_checksum
  | |_ address
  |_ length
```

The checksum calculation of S7 records is identical to S0.

S8 - record

With the linker option **–o***filename***:SREC:3**, which is the default, at the end of an S-record file, the linker generates an S8 record, which contains the program start address.

Layout:

'S' '8' <*length_byte*> <*address*> <*checksum_byte*>

Example:

```
S804FF0003F9
  | |     |_checksum
  | |_ address
  |_ length
```

The checksum calculation of S8 records is identical to S0.

S9 – record

With the linker option **–o***filename***:SREC: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.

7.3 INTEL HEX RECORD FORMAT

Intel Hex records describe the hexadecimal object file format for 8-bit, 16-bit and 32-bit microprocessors. The hexadecimal object file is an ASCII representation of an absolute binary object file. There are six different types of records:

- Data Record (8-, 16, or 32-bit formats)
- End of File Record (8-, 16, or 32-bit formats)
- Extended Segment Address Record (16, or 32-bit formats)
- Start Segment Address Record (16, or 32-bit formats)
- Extended Linear Address Record (32-bit format only)
- Start Linear Address Record (32-bit format only)

For the PCP the linker generates records in the 32-bit format (4-byte addresses with linker option **–o***filename***:IHEX**).

General Record Format

In the output file, the record format is:

:	*length*	*offset*	*type*	*content*	*checksum*

Where:

: is the record header.

length is the record length which specifies the number of bytes of the *content* field. This value occupies one byte (two hexadecimal digits). The linker outputs records of 255 bytes (32 hexadecimal digits) or less; that is, *length* is never greater than 0xFF.

offset is the starting load offset specifying an absolute address in memory where the data is to be located when loaded by a tool. This field is two bytes long. This field is only used for Data Records. In other records this field is coded as four ASCII zero characters ('0000').

type is the record type. This value occupies one byte (two hexadecimal digits). The record types are:

Byte Type	Record type
00	Data
01	End of File
02	Extended segment address (not used)
03	Start segment address (not used)
04	Extended linear address (32–bit)
05	Start linear address (32–bit)

content is the information contained in the record. This depends on the record type.

checksum is the record checksum. The linker computes the checksum by first adding the binary representation of the previous bytes (from *length* to *content*). The linker then computes the result of sum modulo 256 and subtracts the remainder from 256 (two's complement). Therefore, the sum of all bytes following the header is zero.

Extended Linear Address Record

The Extended Linear Address Record specifies the two most significant bytes (bits 16–31) of the absolute address of the first data byte in a subsequent Data Record:

:	02	0000	04	*upper_address*	*checksum*

The 32-bit absolute address of a byte in a Data Record is calculated as:

(*address* + *offset* + *index*) modulo 4G

where:

address is the base address, where the two most significant bytes are the *upper_address* and the two least significant bytes are zero.

offset is the 16-bit offset from the Data Record.

index is the index of the data byte within the Data Record (0 for the first byte).

Example:

```
:0200000400FFFB
 | |    | |    |_ checksum
 | |    | |_ upper_address
 | |    |_ type
 | |_ offset
 |_ length
```

Data Record

The Data Record specifies the actual program code and data.

:	*length*	*offset*	00	*data*	*checksum*

The *length* byte specifies the number of *data* bytes. The linker has an option that controls the length of the output buffer for generating Data records. The default buffer length is 32 bytes.

The *offset* is the 16-bit starting load offset. Together with the address specified in the Extended Address Record it specifies an absolute address in memory where the data is to be located when loaded by a tool.

Example:

```
:0F00200000232222754E00754F04AF4FAE4E22C3
 | |    | |                                |_ checksum
 | |    | |_ data
 | |    |_ type
 | |_ offset
 |_ length
```

Start Linear Address Record

The Start Linear Address Record contains the 32-bit program execution start address.

Layout:

:	04	0000	05	*address*	*checksum*

Example:

```
: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
```

CHAPTER 8

LINKER SCRIPT LANGUAGE

TASKING

8 CHAPTER

8.1 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.

8.2 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 8.5, *Semantics of the Architecture Definition* for detailed descriptions of LSL in the architecture definition.

The derivative definition (required)

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

Altium provides LSL descriptions of supported derivatives, along with "SFR files", which provide easy access to registers in I/O sub-systems from C and assembly programs. When you build an ASIC or use a derivative that is not (yet) supported by the TASKING tools, you may have to write a derivative definition.

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 8.6, *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 8.7, *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 8.7.3, *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 8.8, *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
}
```

8.3 SYNTAX OF THE LINKER SCRIPT LANGUAGE

8.3.1 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.

8.3.2 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 '//'.

8.3.3 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
```

8.3.4 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
```

8.3.5 BUILT-IN FUNCTIONS

```
function_call     ::= absolute ( expr )
                    | addressof ( addr_id )
                    | exists ( section_name )
                    | max ( expr , expr )
                    | min ( expr , expr )
                    | sizeof ( size_id )

addr_id           ::= sect : section_name
                    | group : group_name

size_id           ::= sect : section_name
                    | group : group_name
                    | mem : mem_name
```

- Every space, bus, memory, section or group your refer to, must be defined in the LSL file.
- The `addressof()` and `sizeof()` functions with the **`group`** or **`sect`** argument can only be used in the right hand side of an assignment. The `sizeof()` function with the **`mem`** argument can be used anywhere in section layouts.

You can use the following built-in functions in expressions. All functions return a numerical value. This value is a 64-bit signed integer.

absolute()

```
int absolute( expr )
```

Converts the value of *expr* to a positive integer.

```
absolute( "labelA"-"labelB" )
```

addressof()

```
int addressof( addr_id )
```

Returns the address of *addr_id*, which is a named section or group. To get the offset of the section with the name `asect`:

```
addressof( sect: "asect")
```

This function only works in assignments.

exists()

```
int exists( section_name )
```

The function returns 1 if the section *section_name* exists in one or more object file, 0 otherwise. If the section is not present in input object files, but generated from LSL, the result of this function is undefined.

To check whether the section `mysection` exists in one of the object files that is specified to the linker:

```
exists( "mysection" )
```

max()

```
int max( expr, expr )
```

Returns the value of the expression that has the largest value. To get the highest value of two symbols:

```
max( "sym1" , "sym2")
```

min()

```
int min( expr, expr )
```

Returns the value of the expression hat has the smallest value. To get the lowest value of two symbols:

```
min( "sym1" , "sym2")
```

sizeof()

```
int sizeof( size_id )
```

Returns the size of the object (group, section or memory) the identifier refers to. To get the size of the section "asection":

```
sizeof( sect: "asection" )
```

The **`group`** and **`sect`** arguments only works in assignments. The **`mem`** argument can be used anywhere in section layouts.

8.3.6 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.

8.3.7 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
```

8.3.8 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 <[ range ]
                                  <| [ range ]>>=0 >0|1
                    | page
                    | direction = direction
                    | stack_def
                    | heap_def
                    | copy_table_def
                    | start_address
                    | mapping
```

- A *space_def* contains exactly one **id** and one **mau** statement.
- A *space_def* contains at most one **align** statement.
- A *space_def* contains at most one **page_size** statement.
- A *space_def* contains at least one mapping.

```
stack_def         ::= stack stack_name ( stack_heap_descr
                              <, stack_heap_descr >>=0 )
```

- A *stack_def* defines a stack with the *stack_name* as a unique name.

```
heap_def          ::= heap heap_name ( stack_heap_descr
                              <, stack_heap_descr >>=0 )
```

- A *heap_def* defines a heap with the *heap_name* as a unique name.

```
stack_heap_descr  ::= min_size = expr
                    | grows = direction
                    | align = expr
                    | fixed
```

- 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_def    ::= copytable <( copy_table_descr
                              <, copy_table_descr>>=0 )>0|1
```

- A *space_def* contains at most one **copytable** statement.

- Exactly one copy table must be defined in one of the spaces.

```
copy_table_descr  ::= align = expr
                    | copy_unit = expr
                    | dest <space_name>0|1 = space_name
                    | page
```

- 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
```

8.3.9 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.

```
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.
- At least one *core_def* must be present in a *derivative_definition*.

```
core_descr        ::= architecture = arch_name
                      <( argument_list )>0|1
                    | endianness = ( endianness_type
                                <, endianness_type>>=0 )
```

- An *arch_name* refers to a defined core architecture.
- Exactly one **architecture** statement must be present in a *core_def*.

8.3.10 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.

8.3.11 SECTION PLACEMENT DEFINITION

```
section_definition ::= section_layout <space_ref>0|1
                       <( locate_direction )>0|1
                       { <section_statement>>=0 }
```

- A section definition inside a space definition does not have a *space_ref*.

- All global section definitions have a *space_ref*.

```
space_ref          ::= <proc_name>0|1 : <core_name>0|1
                         : space_name
```

- If more than one processor is present, the *proc_name* must be given for a global section layout.
- If the section layout refers to a processor that has more than one core, the *core_name* must be given in the *space_ref*.
- A *proc_name* refers to a defined processor.
- A *core_name* refers to a defined core.
- A *space_name* refers to a defined address space.

```
locate_direction   ::= direction = direction

direction          ::= low_to_high
                     | high_to_low
```

- A section layout contains at most one **direction** statement.
- If you do not specify the **direction** statement, the locate direction of the section layout is **low-to-high**.

```
section_statement
                   ::= simple_section_statement ;
                     | aggregate_section_statement

simple_section_statement
                   ::= assignment
                     | select_section_statement
                     | special_section_statement

assignment         ::= symbol_name assign_op expr

assign_op          ::= =
                     | :=

select_section_statement
                   ::= select <ref_tree>0|1
                              <section_name>0|1
                         <section_selections>0|1
```

- Either a *section_name* or at least one *section_selection* must be defined.

```
section_selections
                   ::= ( section_selection
                         <, section_selection>>=0 )
```

```
section_selection
                    ::= attributes = < <+|-> attribute>>0
```

- **+***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
                                <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).

```
fill_spec           ::= fill = fill_values

fill_values         ::= expr
                      | [ expr <, expr>>=0 ]

aggregate_section_statement
                    ::= { <section_statement>>=0 }
                      | group_descr
                      | if_statement
                      | section_creation_statement

group_descr         ::= group <group_name>0|1
                              <( group_specs )>0|1
                              section_statement
```

- 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

attribute           ::= r    // readable sections
                      | w    // writable sections
                      | x    // executable code sections
                      | i    // initialized sections
                      | s    // scratch sections
                      | b    // blanked (cleared) sections

group_load_address
                    ::= load_addr <= load_or_run_addr>0|1

group_page          ::= page <= expr>0|1
                      | page_size = expr <[ range ]
                                    <| [ range ]>>=0 >0|1

group_run_address   ::= run_addr <= load_or_run_addr>0|1

group_type          ::= clustered
                      | contiguous
                      | ordered
                      | overlay
```

- For *non-contiguous* groups, you can only specify *group_alignment* and *attributes*.
- The **overlay** keyword also sets the **contiguous** property.
- The **clustered** property cannot be set together with **contiguous** or **ordered** on a single group.

```
load_or_run_addr    ::= addr_absolute
                      | addr_range <| addr_range>>=0
```

```
addr_absolute     ::= expr
                    | memory_reference [ expr ]
```

- An absolute address can only be set on *ordered* groups.

```
addr_range        ::= [ expr .. expr ]
                    | memory_reference
                    | memory_reference [ expr .. expr ]
```

- The parent of a group with an *addr_range* or **page** restriction cannot be **ordered**, **contiguous** or **clustered**.

```
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_specs )
                      { <section_statement2>>=0 }

section_specs     ::= section_spec <, section_spec >>=0

section_spec      ::= attributes
                    | fill_spec
                    | size = expr
                    | blocksize = expr
                    | overflow = section_name

section_statement2
                  ::= select_section_statement ;
                    | group_descr2
                    | { <section_statement2>>=0 }

group_descr2      ::= group <group_name>0|1
                            ( group_specs2 )
                            section_statement2

group_specs2      ::= group_spec2 <, group_spec2 >>=0

group_spec2       ::= group_alignment
                    | attributes
                    | load_addr
```

8.4 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.

8.5 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
   page
   direction        low_to_high  high_to_low
   stack
      min_size
      grows         low_to_high  high_to_low
      align
      fixed
   heap
      min_size
      grows         low_to_high  high_to_low
      align
      fixed
   copytable
      align
      copy_unit
      dest
      page
   start_address
      run_addr
      symbol
   map

   map
      dest          bus  space
      dest_dbits
      dest_offset
      size
      src_dbits
      src_offset
```

8.5.1 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
}
```

8.5.2 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 8.5.4, *Mappings*.

```
bus bus_name
{
     mau = 8;
     width = 8;
     map ( map_description );
}
```

8.5.3 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 alignment and page size in MAUs for the address space. It must be a power of 2. The default value is 1. If one or more page ranges are supplied the supplied value only sets the page alignment. The ranges specify the available space in each page, as offsets to the page start, which is aligned at the page alignment.

  See also the **page** keyword in subsection *Locating a group* in section 8.8.2, *Creating and Locating Groups of Sections*.

- With the optional **direction** field you can specify how all sections in this space should be located. This can be either from **low_to_high** addresses (this is the default) or from **high_to_low** addresses.
- The **map** keyword specifies how this address space maps onto an internal bus or onto another address space. Mappings are described in section 8.5.4, *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 8.8.3, *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 8.8.3, *Creating or Modifying Special Sections*.

See section 8.8, *Semantics of the Section Layout Definition* for information on creating and placing stack sections.

Copy tables

- The **`copytable`** keyword defines a copy table in the address space. The content of the copy table is created by the linker and contains the start address and size of all sections that should be initialized by the startup code. You must define exactly one copy table in one of the address spaces (for a core).

  Optionally you can specify an alignment for the copy table with the argument **`align`**. This alignment must be equal or larger than the alignment that you specify for the address space itself. If smaller, the alignment for the address space is used.

  The **`copy_unit`** argument specifies the size in MAUs of information chunks that are copied. If you do not specify the copy unit, the MAU size of the address space itself is used.

  The **`dest`** argument specifies the destination address space that the code uses for the copy table. The linker uses this information to generate the correct addresses in the copy table. The memory into where the sections must be copied at run-time, must be accessible from this destination space.

  Sections generated for the copy table may get a page restriction with the address space's page size, by adding the **`page`** argument.

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 );
}
```

8.5.4 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.

8.6 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
```

8.6.1 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
}
```

8.6.2 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);
}
```

8.6.3 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 8.7.3, *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 8.8.2, *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 8.5.4, *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 8.5.2, *Defining Internal Buses*.

8.7 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
```

8.7.1 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
}
```

8.7.2 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);
}
```

8.7.3 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 8.6.3, *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 8.5.2, *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.

8.8 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
   page_size
select
   ref_tree
heap
   size
stack
   size
reserved
   size
   attributes     r w x
   fill
   alloc_allowed absolute
copytable
section
   size
   blocksize
   attributes     r w x
   fill
   overflow

if
else
```

8.8.1 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.

8.8.2 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 8.8.3, *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.

- With the **ref_tree** field you can select a group of related sections. The relation between sections is often expressed by means of references. By selecting just the 'root' of tree, the complete tree is selected. This is for example useful to locate a group of related sections in special memory (e.g. fast memory). The (referenced) sections must meet the following conditions in order to be selected:
  1. The sections are within the section layout's address space
  2. The sections match the specified attributes
  3. The sections have no absolute restriction (as is the case for all wildcard selections)

For example, to select the code sections referenced from `foo1`:

```
group refgrp (ordered, contiguous, run_addr=mem:ext_c)
{
    select ref_tree "foo1" (attributes=+x);
}
```

If section `foo1` references `foo2` and `foo2` references `foo3`, then all these sections are selected by the selection shown above.

Locating a group

```
group group_name ( group_specifications )
{
   section_statements
}
```

With the *group_specifications* you actually define how the linker must locate the group. You can roughly define three things: 1) assign properties to the group like alignment and read/write attributes, 2) define the mutual order in the address space for sections in the group and 3) restrict the possible addresses for the sections in a group.

The linker creates labels that allow you to refer to the begin and end address of a group from within the application software. Labels **_lc_gb_***group_name* and **_lc_ge_***group_name* mark the begin and end of the group respectively, where the begin is the lowest address used within this group and the end is the highest address used. Notice that a group not necessarily occupies all memory between begin and end address. The given label refers to where the section is located at run-time (versus load-time).

1. Assign properties to the group like alignment and read/write attributes. These properties are assigned to all sections in the group (and subgroups) and override the attributes of the input sections.
   - The **align** field tells the linker to align all sections in the group and the group as a whole according to the align value. By default the linker uses the largest alignment constraint of either the input sections or the alignment of the address space.
   - The **attributes** field tells the linker to assign one or more attributes to all sections in the group. This overrules the default attributes. By default the linker uses the attributes of the input sections. You can set the **r**, **w**, or **rw** attributes and you can switch between the **b** and **s** attributes.

2. Define the mutual order of the sections in the group.
   By default, a group is *unrestricted* which means that the linker has total freedom to place the sections of the group in the address space.

   - The **ordered** keyword tells the linker to locate the sections in the same order in the address space as they appear in the group (but not necessarily adjacent).

     Suppose you have an ordered group that contains the sections 'A', 'B' and 'C'. By default the linker places the sections in the address space like 'A' - 'B' - 'C', where section 'A' gets the lowest possible address. With **direction=high_to_low** in the **section_layout** space properties, the linker places the sections in the address space like 'C' - 'B' - 'A', where section 'A' gets the highest possible address.

   - The **contiguous** keyword tells the linker to locate the sections in the group in a single address range. Within a contiguous group the input sections are located in arbitrary order, however the group occupies one contiguous range of memory. Due to alignment of sections there can be 'alignment gaps' between the sections.

     When you define a group that is both **ordered** and **contiguous**, this is called a *sequential* group. In a sequential group the linker places sections in the same order in the address space as they appear in the group and it occupies a contiguous range of memory.

   - The **clustered** keyword tells the linker to locate the sections in the group in a number of *contiguous* blocks. It tries to keep the number of these blocks to a minimum. If enough memory is available, the group will be located as if it was specified as **contiguous**. Otherwise, it gets split into two or more blocks.

     If a contiguous or clustered group contains *alignment gaps*, the linker can locate sections that are not part of the group in these gaps. To prevent this, you can use the **fill** keyword. If the group is located in RAM, the gaps are treated as reserved (scratch) space. If the group is located in ROM, the alignment gaps are filled with zeros by default. You can however change the fill pattern by specifying a bit pattern. The result of the expression, or list of expressions, is used as values to write to memory, each in MAU.

   - The **overlay** keyword tells the linker to overlay the sections in the group. The linker places all sections in the address space using a contiguous range of addresses. (Thus an overlay group is automatically also a contiguous group.) To overlay the sections, all sections in the overlay group share the same run-time address.

For each input section within the overlay the linker automatically defines two symbols. The symbol **_lc_cb_***section_name* is defined as the load-time start address of the section. The symbol **_lc_ce_***section_name* is defined as the load-time end address of the section. C (or assembly) code may be used to copy the overlaid sections.

If sections in the overlay group contain references between groups, the linker reports an error. The keyword **allow_cross_references** tells the linker to accept cross-references. Normally, it does not make sense to have references between sections that are overlaid.

```
group ovl (overlay)
{
    group a
    {
        select "my_ovl_p1";
        select "my_ovl_p2";
    }
    group b
    {
        select "my_ovl_q1";
    }
}
```

It may be possible that one of the sections in the overlay group already has been defined in another group where it received a load-time address. In this case the linker does not overrule this load-time address and excludes the section from the overlay group.

3. Restrict the possible addresses for the sections in a group.
   The load-time address specifies where the group's elements are loaded in memory at download time. The run-time address specifies where sections are located at run-time, that is when the program is executing. If you do not explicitly restrict the address in the LSL file, the linker assigns addresses to the sections based on the restrictions relative to other sections in the LSL file and section alignments. The program is responsible for copying overlay sections at appropriate moment from its load-time location to its run-time location (this is typically done by the startup code).

- The **run_addr** keyword defines the run-time address. If the run-time location of a group is set explicitly, the given order between groups specify whether the run-time address propagates to the parent group or not. The location of the sections a group can be restricted either to a single absolute address, or to a number of address ranges. 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.

- With the **page_size** keyword you can override the page alignment and size set on the address space. See also the **page_size** keyword in section 8.5.3, *Defining Address Spaces*.

```
group (page, ... )
group (page = 3, ...)
```

8.8.3 CREATING OR MODIFYING SPECIAL SECTIONS

Instead of selecting sections, you can also create a reserved section or an output section or modify special sections like a stack or a heap. Because you cannot define these sections in the input files, you must use the linker to create them.

Stack

- The keyword **stack** tells the linker to reserve memory for the stack. The name for the stack section refers to the stack as defined in the architecture definition. If no name was specified in the architecture definition, the default name is **stack**.

With the keyword **size** you can specify the size for the stack. If the **size** is not specified, the linker uses the size given by the **min_size** argument as defined for the stack in the architecture definition. Normally the linker automatically tries to maximize the size, unless you specified the keyword **fixed**.

```
group ( ... )
{
   stack "mystack" ( size = 2k );
}
```

The linker creates two labels to mark the begin and end of the stack, **_lc_ub_***stack_name* for the begin of the stack and **_lc_ue_***stack_name* for the end of the stack. The linker allocates space for the stack when there is a reference to either of the labels.

See also the **stack** keyword in section 8.5.3, *Defining Address Spaces*.

Heap

- The keyword **heap** tells the linker to reserve a dynamic memory range for the `malloc()` function. Each heap section has a name. With the keyword **size** you can change the size for the heap. If the **size** is not specified, the linker uses the size given by the **min_size** argument as defined for the heap in the architecture definition. Normally the linker automatically tries to maximize the size, unless you specified the keyword **fixed**.

  ```
  group ( ... )
  {
     heap "myheap" ( size = 2k );
  }
  ```

  The linker creates two labels to mark the begin and end of the heap, **_lc_ub_***heap_name* for the begin of the heap and **_lc_ue_***heap_name* for the end of the heap. The linker allocates space for the heap when a reference to either of the section labels exists in one of the input object files.

Reserved section

- The keyword **reserved** tells the linker to create an area or section of a given size. The linker will not locate any other sections in the memory occupied by a reserved section, with some exceptions. Each reserved section has a name. With the keyword **size** you can specify a size for a given reserved area or section.

```
group ( ... )
{
   reserved "myreserved" ( size = 2k );
}
```

The optional **fill** field contains a bit pattern that the linker writes to all memory addresses that remain unoccupied during the locate process. The result of the expression, or list of expressions, is used as values to write to memory, each in MAU. The first MAU of the fill pattern is always the first MAU in the section.

By default, no sections can overlap with a reserved section. With **alloc_allowed=absolute** sections that are located at an absolute address due to an absolute group restriction can overlap a reserved section.

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. You can use groups inside output sections, but you can only set the **align**, **attributes** and **load_addr** attributes.

  The **fill** field contains a bit pattern that the linker writes to all unused space in the output section. When all input sections have an image (code/data) you must specify a fill pattern. If you do not specify a fill pattern, all input sections must be scratch sections. The fill pattern is aligned at the start of the output section.

  As with a reserved section you can use the **attributes** field to set the access type of the output section.

```
group ( ... )
{
   section "myoutput" ( size = 4k, attributes = rw,
                        fill = 0xaa )
   {
      select "myinput1";
      select "myinput2";
   }
}
```

  The available room for input sections is determined by the **size**, **blocksize** and **overflow** fields. With the keyword **size** you specify the fixed size of the output section. Input sections are placed from output section start towards higher addresses (offsets). When the end of the output section is reached and one or more input sections are not yet placed, an error is emitted. If however, the **overflow** field is set to another output section, remaining sections are located as if they were selected for the overflow output section.

```
group ( ... )
{
  section "tsk1_data" (size=4k, attributes=rw, fill=0,
                       overflow = "overflow_data")
  {
          select ".data.tsk1.*"
  }
  section "tsk2_data" (size=4k, attributes=rw, fill=0,
                       overflow = "overflow_data")
  {
          select ".data.tsk2.*"
  }
  section "overflow_data" (size=4k, attributes=rx,
                           fill=0)
  {
  }
}
```

With the keyword **blocksize** , the size of the output section will adapt to the size of its content. For example:

```
group flash_area (run_addr = 0x10000)
{
   section "flash_code" (blocksize=4k, attributes=rx,
                         fill=0)
   {
     select "*.flash";
   }
}
```

If the content of the section is 1 mau, the size will be 4k, if the content is 11k, the section will be 12k, etc. If you use **size** in combination with **blocksize**, the **size** value is used as default (minimal) size for this section. If it is omitted, the default size will be of **blocksize**. It is not allowed to omit both **size** and **blocksize** from the section definition.

The linker creates two labels to mark the begin and end of the section, **_lc_ub_***name* for the start, and **_lc_ue_***name* for the end of the output section.

Copy table

- The keyword **copytable** tells the linker to select a section that is used as *copy table*. The content of the copy table is created by the linker. It contains the start address and length of all sections that should be initialized by the startup code.

  The linker creates two labels to mark the begin and end of the section, **_lc_ub_table** for the 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.

8.8.4 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
}
```

8.8.5 CONDITIONAL GROUP STATEMENTS

Within a group, you can conditionally select sections or create special sections.

- With the **if** keyword you can specify a condition. The succeeding section statement is executed if the condition evaluates to TRUE (1).
- The optional **else** keyword is followed by a section statement which is executed in case the if-condition evaluates to FALSE (0).

```
group ( ... )
{
   if ( exists( "mysection" ) )
      select "mysection";
   else
      reserved "myreserved" ( size=2k );
}
```

CHAPTER 9

MISRA-C RULES

TASKING

CHAPTER 6

9.1 MISRA-C:1998

This section lists all supported and unsupported MISRA-C:1998 rules.

See also section 4.7, *C Code Checking: MISRA-C*, in Chapter *Using the Compiler* of the *User's Manual*.

A number of MISRA-C rules leave room for interpretation. Other rules can only be checked in a limited way. In such cases the implementation decisions and possible restrictions for these rules are listed.

x means that the rule is not supported by the TASKING C compiler.
(R) is a required rule, (A) is an advisory rule.

	1.	(R)	The code shall conform to standard C, without language extensions
x	2.	(A)	Other languages should only be used with an interface standard
	3.	(A)	Inline assembly is only allowed in dedicated C functions
x	4.	(A)	Provision should be made for appropriate run-time checking
	5.	(R)	Only use characters and escape sequences defined by ISO C
x	6.	(R)	Character values shall be restricted to a subset of ISO 106460-1
	7.	(R)	Trigraphs shall not be used
	8.	(R)	Multibyte characters and wide string literals shall not be used
	9.	(R)	Comments shall not be nested
	10.	(A)	Sections of code should not be "commented out" In general, it is not possible to decide whether a piece of comment is C code that is commented out, or just some pseudo code. Instead, the following heuristics are used to detect possible C code inside a comment: - a line ends with ';', or - a line starts with '}', possibly preceded by white space
	11.	(R)	Identifiers shall not rely on significance of more than 31 characters
	12.	(A)	The same identifier shall not be used in multiple name spaces

	13.	(A)	Specific-length typedefs should be used instead of the basic types
	14.	(R)	Use 'unsigned char' or 'signed char' instead of plain 'char'
x	15.	(A)	Floating point implementations should comply with a standard
	16.	(R)	The bit representation of floating point numbers shall not be used A violation is reported when a pointer to a floating point type is converted to a pointer to an integer type.
	17.	(R)	"typedef" names shall not be reused
	18.	(A)	Numeric constants should be suffixed to indicate type A violation is reported when the value of the constant is outside the range indicated by the suffixes, if any.
	19.	(R)	Octal constants (other than zero) shall not be used
	20.	(R)	All object and function identifiers shall be declared before use
	21.	(R)	Identifiers shall not hide identifiers in an outer scope
	22.	(A)	Declarations should be at function scope where possible
x	23.	(A)	All declarations at file scope should be static where possible
	24.	(R)	Identifiers shall not have both internal and external linkage
x	25.	(R)	Identifiers with external linkage shall have exactly one definition
	26.	(R)	Multiple declarations for objects or functions shall be compatible
x	27.	(A)	External objects should not be declared in more than one file
	28.	(A)	The "register" storage class specifier should not be used
	29.	(R)	The use of a tag shall agree with its declaration
	30.	(R)	All automatics shall be initialized before being used This rule is checked using worst-case assumptions. This means that violations are reported not only for variables that are guaranteed to be uninitialized, but also for variables that are uninitialized on some execution paths.
	31.	(R)	Braces shall be used in the initialization of arrays and structures

32. (R) Only the first, or all enumeration constants may be initialized

33. (R) The right hand operand of && or || shall not contain side effects

34. (R) The operands of a logical && or || shall be primary expressions

35. (R) Assignment operators shall not be used in Boolean expressions

36. (A) Logical operators should not be confused with bitwise operators

37. (R) Bitwise operations shall not be performed on signed integers

38. (R) A shift count shall be between 0 and the operand width minus 1

    This violation will only be checked when the shift count evaluates to a constant value at compile time.

39. (R) The unary minus shall not be applied to an unsigned expression

40. (A) "sizeof" should not be used on expressions with side effects

x 41. (A) The implementation of integer division should be documented

42. (R) The comma operator shall only be used in a "for" condition

43. (R) Don't use implicit conversions which may result in information loss

44. (A) Redundant explicit casts should not be used

45. (R) Type casting from any type to or from pointers shall not be used

46. (R) The value of an expression shall be evaluation order independent

    This rule is checked using worst-case assumptions. This means that a violation will be reported when a possible alias may cause the result of an expression to be evaluation order dependent.

47. (A) No dependence should be placed on operator precedence rules

48. (A) Mixed arithmetic should use explicit casting

49. (A) Tests of a (non-Boolean) value against 0 should be made explicit
50. (R) F.P. variables shall not be tested for exact equality or inequality
51. (A) Constant unsigned integer expressions should not wrap-around
52. (R) There shall be no unreachable code
53. (R) All non-null statements shall have a side-effect
54. (R) A null statement shall only occur on a line by itself
55. (A) Labels should not be used
56. (R) The ”goto” statement shall not be used
57. (R) The ”continue” statement shall not be used
58. (R) The ”break” statement shall not be used (except in a ”switch”)
59. (R) An ”if” or loop body shall always be enclosed in braces
60. (A) All ”if”, ”else if” constructs should contain a final ”else”
61. (R) Every non-empty ”case” clause shall be terminated with a ”break”
62. (R) All ”switch” statements should contain a final ”default” case
63. (A) A ”switch” expression should not represent a Boolean case
64. (R) Every ”switch” shall have at least one ”case”
65. (R) Floating point variables shall not be used as loop counters
66. (A) A ”for” should only contain expressions concerning loop control

    A violation is reported when the loop initialization or loop update expression modifies an object that is not referenced in the loop test.
67. (A) Iterator variables should not be modified in a ”for” loop
68. (R) Functions shall always be declared at file scope
69. (R) Functions with variable number of arguments shall not be used

70. (R) Functions shall not call themselves, either directly or indirectly

    A violation will be reported for direct or indirect recursive function calls in the source file being checked. Recursion via functions in other source files, or recursion via function pointers is not detected.

71. (R) Function prototypes shall be visible at the definition and call

72. (R) The function prototype of the declaration shall match the definition

73. (R) Identifiers shall be given for all prototype parameters or for none

74. (R) Parameter identifiers shall be identical for declaration/definition

75. (R) Every function shall have an explicit return type

76. (R) Functions with no parameters shall have a "void" parameter list

77. (R) An actual parameter type shall be compatible with the prototype

78. (R) The number of actual parameters shall match the prototype

79. (R) The values returned by "void" functions shall not be used

80. (R) Void expressions shall not be passed as function parameters

81. (A) "const" should be used for reference parameters not modified

82. (A) A function should have a single point of exit

83. (R) Every exit point shall have a "return" of the declared return type

84. (R) For "void" functions, "return" shall not have an expression

85. (A) Function calls with no parameters should have empty parentheses

86. (A) If a function returns error information, it should be tested

    A violation is reported when the return value of a function is ignored.

87. (R) #include shall only be preceded by other directives or comments

88. (R) Non-standard characters shall not occur in #include directives

89. (R) #include shall be followed by either <filename> or "filename"

90. (R) Plain macros shall only be used for constants/qualifiers/specifiers

91. (R) Macros shall not be #define'd and #undef'd within a block

92. (A) #undef should not be used

93. (A) A function should be used in preference to a function-like macro

94. (R) A function-like macro shall not be used without all arguments

95. (R) Macro arguments shall not contain pre-preprocessing directives

    A violation is reported when the first token of an actual macro argument is '#'.

96. (R) Macro definitions/parameters should be enclosed in parentheses

97. (A) Don't use undefined identifiers in pre-processing directives

98. (R) A macro definition shall contain at most one # or ## operator

99. (R) All uses of the #pragma directive shall be documented

    This rule is really a documentation issue. The compiler will flag all #pragma directives as violations.

100. (R) "defined" shall only be used in one of the two standard forms

101. (A) Pointer arithmetic should not be used

102. (A) No more than 2 levels of pointer indirection should be used

    A violation is reported when a pointer with three or more levels of indirection is declared.

103. (R) No relational operators between pointers to different objects

    In general, checking whether two pointers point to the same object is impossible. The compiler will only report a violation for a relational operation with incompatible pointer types.

104. (R) Non-constant pointers to functions shall not be used

105. (R) Functions assigned to the same pointer shall be of identical type

	106.	(R)	Automatic address may not be assigned to a longer lived object
	107.	(R)	The null pointer shall not be de-referenced A violation is reported for every pointer dereference that is not guarded by a NULL pointer test.
	108.	(R)	All struct/union members shall be fully specified
	109.	(R)	Overlapping variable storage shall not be used A violation is reported for every 'union' declaration.
	110.	(R)	Unions shall not be used to access the sub-parts of larger types A violation is reported for a 'union' containing a 'struct' member.
	111.	(R)	Bit fields shall have type "unsigned int" or "signed int"
	112.	(R)	Bit fields of type "signed int" shall be at least 2 bits long
	113.	(R)	All struct/union members shall be named
	114.	(R)	Reserved and standard library names shall not be redefined
	115.	(R)	Standard library function names shall not be reused
x	116.	(R)	Production libraries shall comply with the MISRA-C restrictions
x	117.	(R)	The validity of library function parameters shall be checked
	118.	(R)	Dynamic heap memory allocation shall not be used
	119.	(R)	The error indicator "errno" shall not be used
	120.	(R)	The macro "offsetof" shall not be used
	121.	(R)	<locale.h> and the "setlocale" function shall not be used
	122.	(R)	The "setjmp" and "longjmp" functions shall not be used
	123.	(R)	The signal handling facilities of <signal.h> shall not be used
	124.	(R)	The <stdio.h> library shall not be used in production code
	125.	(R)	The functions atof/atoi/atol shall not be used
	126.	(R)	The functions abort/exit/getenv/system shall not be used
	127.	(R)	The time handling functions of library <time.h> shall not be used

9.2 MISRA-C:2004

This section lists all supported and unsupported MISRA-C:2004 rules.

See also section 4.7, *C Code Checking: MISRA-C*, in Chapter *Using the Compiler* of the *User's Manual*.

A number of MISRA-C rules leave room for interpretation. Other rules can only be checked in a limited way. In such cases the implementation decisions and possible restrictions for these rules are listed.

x means that the rule is not supported by the TASKING C compiler.
(R) is a required rule, (A) is an advisory rule.

Environment

	1.1	(R)	All code shall conform to ISO 9899:1990 "Programming languages - C", amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996.
	1.2	(R)	No reliance shall be placed on undefined or unspecified behavior.
x	1.3	(R)	Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the languages/compilers/assemblers conform.
x	1.4	(R)	The compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.
x	1.5	(A)	Floating-point implementations should comply with a defined floating-point standard.

Language extensions

	2.1	(R)	Assembly language shall be encapsulated and isolated.
	2.2	(R)	Source code shall only use `/* ... */` style comments.

2.3 (R) The character sequence `/*` shall not be used within a comment.

2.4 (A) Sections of code should not be "commented out".

In general, it is not possible to decide whether a piece of comment is C code that is commented out, or just some pseudo code. Instead, the following heuristics are used to detect possible C code inside a comment:

- a line ends with ';', or

- a line starts with '}', possibly preceded by white space

Documentation

x 3.1 (R) All usage of implementation-defined behavior shall be documented.

x 3.2 (R) The character set and the corresponding encoding shall be documented.

x 3.3 (A) The implementation of integer division in the chosen compiler should be determined, documented and taken into account.

3.4 (R) All uses of the `#pragma` directive shall be documented and explained.

This rule is really a documentation issue. The compiler will flag all `#pragma` directives as violations.

3.5 (R) The implementation-defined behavior and packing of bit fields shall be documented if being relied upon.

x 3.6 (R) All libraries used in production code shall be written to comply with the provisions of this document, and shall have been subject to appropriate validation.

Character sets

4.1 (R) Only those escape sequences that are defined in the ISO C standard shall be used.

4.2 (R) Trigraphs shall not be used.

Identifiers

	5.1	(R)	Identifiers (internal and external) shall not rely on the significance of more than 31 characters.
	5.2	(R)	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
	5.3	(R)	A `typedef` name shall be a unique identifier.
	5.4	(R)	A tag name shall be a unique identifier.
x	5.5	(A)	No object or function identifier with static storage duration should be reused.
	5.6	(A)	No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure and union member names.
x	5.7	(A)	No identifier name should be reused.

Types

	6.1	(R)	The plain `char` type shall be used only for storage and use of character values.
x	6.2	(R)	`signed` and `unsigned char` type shall be used only for the storage and use of numeric values.
	6.3	(A)	`typedef`s that indicate size and signedness should be used in place of the basic types.
	6.4	(R)	Bit fields shall only be defined to be of type `unsigned int` or `signed int`.
	6.5	(R)	Bit fields of type `signed int` shall be at least 2 bits long.

Constants

	7.1	(R)	Octal constants (other than zero) and octal escape sequences shall not be used.

Declarations and definitions

	8.1	(R)	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.
	8.2	(R)	Whenever an object or function is declared or defined, its type shall be explicitly stated.

	8.3	(R)	For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.
	8.4	(R)	If objects or functions are declared more than once their types shall be compatible.
	8.5	(R)	There shall be no definitions of objects or functions in a header file.
	8.6	(R)	Functions shall be declared at file scope.
	8.7	(R)	Objects shall be defined at block scope if they are only accessed from within a single function.
x	8.8	(R)	An external object or function shall be declared in one and only one file.
x	8.9	(R)	An identifier with external linkage shall have exactly one external definition.
x	8.10	(R)	All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.
	8.11	(R)	The `static` storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
	8.12	(R)	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.

Initialization

	9.1	(R)	All automatic variables shall have been assigned a value before being used. This rule is checked using worst-case assumptions. This means that violations are reported not only for variables that are guaranteed to be uninitialized, but also for variables that are uninitialized on some execution paths.
	9.2	(R)	Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.
	9.3	(R)	In an enumerator list, the "=" construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.

Arithmetic type conversions

10.1 (R) The value of an expression of integer type shall not be implicitly converted to a different underlying type if:
a) it is not a conversion to a wider integer type of the same signedness, or
b) the expression is complex, or
c) the expression is not constant and is a function argument, or
d) the expression is not constant and is a return expression.

10.2 (R) The value of an expression of floating type shall not be implicitly converted to a different type if:
a) it is not a conversion to a wider floating type, or
b) the expression is complex, or
c) the expression is a function argument, or
d) the expression is a return expression.

10.3 (R) The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression.

10.4 (R) The value of a complex expression of floating type may only be cast to a narrower floating type.

10.5 (R) If the bitwise operators `~` and `<<` are applied to an operand of underlying type `unsigned char` or `unsigned short`, the result shall be immediately cast to the underlying type of the operand.

10.6 (R) A "U" suffix shall be applied to all constants of `unsigned` type.

Pointer type conversions

11.1 (R) Conversions shall not be performed between a pointer to a function and any type other than an integral type.

11.2 (R) Conversions shall not be performed between a pointer to object and any type other than an integral type, another pointer to object type or a pointer to void.

11.3 (A) A cast should not be performed between a pointer type and an integral type.

11.4 (A) A cast should not be performed between a pointer to object type and a different pointer to object type.

11.5 (R) A cast shall not be performed that removes any `const` or `volatile` qualification from the type addressed by a pointer.

Expressions

12.1 (A) Limited dependence should be placed on C's operator precedence rules in expressions.

12.2 (R) The value of an expression shall be the same under any order of evaluation that the standard permits.

This rule is checked using worst-case assumptions. This means that a violation will be reported when a possible alias may cause the result of an expression to be evaluation order dependent.

12.3 (R) The `sizeof` operator shall not be used on expressions that contain side effects.

12.4 (R) The right-hand operand of a logical `&&` or || operator shall not contain side effects.

12.5 (R) The operands of a logical `&&` or || shall be *primary-expressions*.

12.6 (A) The operands of logical operators (`&&`, || and `!`) should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to operators other than (`&&`, || and `!`).

12.7 (R) Bitwise operators shall not be applied to operands whose underlying type is signed.

12.8 (R) The right-hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left-hand operand.

This violation will only be checked when the shift count evaluates to a constant value at compile time.

12.9 (R) The unary minus operator shall not be applied to an expression whose underlying type is unsigned.

12.10 (R) The comma operator shall not be used.

12.11 (A) Evaluation of constant unsigned integer expressions should not lead to wrap-around.

12.12 (R) The underlying bit representations of floating-point values shall not be used.

A violation is reported when a pointer to a floating point type is converted to a pointer to an integer type.

12.13 (A) The increment (`++`) and decrement (`--`) operators should not be mixed with other operators in an expression.

Control statement expressions

13.1 (R) Assignment operators shall not be used in expressions that yield a Boolean value.

13.2 (A) Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.

13.3 (R) Floating-point expressions shall not be tested for equality or inequality.

13.4 (R) The controlling expression of a `for` statement shall not contain any objects of floating type.

13.5 (R) The three expressions of a `for` statement shall be concerned only with loop control.

A violation is reported when the loop initialization or loop update expression modifies an object that is not referenced in the loop test.

13.6 (R) Numeric variables being used within a `for` loop for iteration counting shall not be modified in the body of the loop.

13.7 (R) Boolean operations whose results are invariant shall not be permitted.

Control flow

14.1 (R) There shall be no unreachable code.

14.2 (R) All non-null statements shall either:
a) have at least one side effect however executed, or
b) cause control flow to change.

14.3 (R) Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment provided that the first character following the null statement is a white-space character.

14.4 (R) The `goto` statement shall not be used.

14.5 (R) The `continue` statement shall not be used.

14.6 (R) For any iteration statement there shall be at most one break statement used for loop termination.

14.7 (R) A function shall have a single point of exit at the end of the function.

14.8 (R) The statement forming the body of a `switch`, `while`, `do ... while` or `for` statement be a compound statement.

14.9 (R) An `if (`*`expression`*`)` construct shall be followed by a compound statement. The `else` keyword shall be followed by either a compound statement, or another `if` statement.

14.10 (R) All `if ... else if` constructs shall be terminated with an `else` clause.

Switch statements

15.1 (R) A switch label shall only be used when the most closely-enclosing compound statement is the body of a `switch` statement.

15.2 (R) An unconditional `break` statement shall terminate every non-empty `switch` clause.

15.3 (R) The final clause of a switch statement shall be the `default` clause.

15.4 (R) A `switch` expression shall not represent a value that is effectively Boolean.

15.5 (R) Every `switch` statement shall have at least one `case` clause.

Functions

16.1 (R) Functions shall not be defined with variable numbers of arguments.

16.2 (R) Functions shall not call themselves, either directly or indirectly.

A violation will be reported for direct or indirect recursive function calls in the source file being checked. Recursion via functions in other source files, or recursion via function pointers is not detected.

16.3 (R) Identifiers shall be given for all of the parameters in a function prototype declaration.

16.4 (R) The identifiers used in the declaration and definition of a function shall be identical.

16.5 (R) Functions with no parameters shall be declared with parameter type `void`.

16.6 (R) The number of arguments passed to a function shall match the number of parameters.

16.7 (A) A pointer parameter in a function prototype should be declared as pointer to `const` if the pointer is not used to modify the addressed object.

16.8 (R) All exit paths from a function with non-void return type shall have an explicit `return` statement with an expression.

16.9 (R) A function identifier shall only be used with either a preceding `&`, or with a parenthesized parameter list, which may be empty.

16.10 (R) If a function returns error information, then that error information shall be tested.

A violation is reported when the return value of a function is ignored.

Pointers and arrays

x 17.1 (R) Pointer arithmetic shall only be applied to pointers that address an array or array element.

x 17.2 (R) Pointer subtraction shall only be applied to pointers that address elements of the same array.

17.3 (R) `>`, `>=`, `<`, `<=` shall not be applied to pointer types except where they point to the same array.

In general, checking whether two pointers point to the same object is impossible. The compiler will only report a violation for a relational operation with incompatible pointer types.

17.4 (R) Array indexing shall be the only allowed form of pointer arithmetic.

17.5 (A) The declaration of objects should contain no more than 2 levels of pointer indirection.

A violation is reported when a pointer with three or more levels of indirection is declared.

17.6 (R) The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

Structures and unions

18.1 (R) All structure or union types shall be complete at the end of a translation unit.

18.2 (R) An object shall not be assigned to an overlapping object.

x 18.3 (R) An area of memory shall not be reused for unrelated purposes.

18.4 (R) Unions shall not be used.

Preprocessing directives

19.1 (A) `#include` statements in a file should only be preceded by other preprocessor directives or comments.

19.2 (A) Non-standard characters should not occur in header file names in `#include` directives.

x 19.3 (R) The `#include` directive shall be followed by either a <filename> or "filename" sequence.

19.4 (R) C macros shall only expand to a braced initializer, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct.

19.5 (R) Macros shall not be `#define`'d or `#undef`'d within a block.

19.6 (R) `#undef` shall not be used.

19.7 (A) A function should be used in preference to a function-like macro.

19.8 (R) A function-like macro shall not be invoked without all of its arguments.

19.9 (R) Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.

A violation is reported when the first token of an actual macro argument is '#'.

19.10 (R) In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##.

19.11 (R) All macro identifiers in preprocessor directives shall be defined before use, except in `#ifdef` and `#ifndef` preprocessor directives and the `defined()` operator.

19.12 (R) There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.

19.13 (A) The # and ## preprocessor operators should not be used.

19.14 (R) The `defined` preprocessor operator shall only be used in one of the two standard forms.

19.15 (R) Precautions shall be taken in order to prevent the contents of a header file being included twice.

19.16 (R) Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.

19.17 (R) All `#else`, `#elif` and `#endif` preprocessor directives shall reside in the same file as the `#if` or `#ifdef` directive to which they are related.

Standard libraries

20.1 (R) Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined.

20.2 (R) The names of standard library macros, objects and functions shall not be reused.

x 20.3 (R) The validity of values passed to library functions shall be checked.

20.4 (R) Dynamic heap memory allocation shall not be used.

20.5 (R) The error indicator `errno` shall not be used.

20.6 (R) The macro `offsetof`, in library `<stddef.h>`, shall not be used.

20.7 (R) The `setjmp` macro and the `longjmp` function shall not be used.

20.8 (R) The signal handling facilities of `<signal.h>` shall not be used.

20.9 (R) The input/output library `<stdio.h>` shall not be used in production code.

20.10 (R) The library functions `atof`, `atoi` and `atol` from library `<stdlib.h>` shall not be used.

20.11 (R) The library functions `abort`, `exit`, `getenv` and `system` from library `<stdlib.h>` shall not be used.

20.12 (R) The time handling functions of library `<time.h>` shall not be used.

Run-time failures

x 21.1 (R) Minimization of run-time failures shall be ensured by the use of at least one of:
a) static analysis tools/techniques;
b) dynamic analysis tools/techniques;
c) explicit coding of checks to handle run-time faults.

INDEX

INDEX

TASKING

INDEX

Symbols

A

B

C

INDEX

D

E

F

G

H

INDEX

INDEX

M

N

O

P

INDEX

Q

R

S

T

U

V

W

X