```c
FILE* sfile;
int count = 0;

sfile = fopen(file);

if (sfile == NULL) {
    return -1;
}

while (!)
{
    char c;
    c = fgetc(sfile);
    if (c == EOF)
    {
        break;
    }
    else
    {
    }
}

return count;
```
The information in this document has been carefully reviewed and is believed to be accurate and reliable. However, Altium assumes no liabilities for inaccuracies in this document. Furthermore, the delivery of this information does not convey to the recipient any license to use or copy the software or documentation, except as provided in an executed license agreement covering the software and documentation.

Altium reserves the right to change specifications embodied in this document without prior notice.
**PCP C LANGUAGE**

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Introduction</td>
<td>1-3</td>
</tr>
<tr>
<td>1.2</td>
<td>Data Types</td>
<td>1-4</td>
</tr>
<tr>
<td>1.3</td>
<td>Keywords</td>
<td>1-5</td>
</tr>
<tr>
<td>1.4</td>
<td>Function Qualifiers</td>
<td>1-7</td>
</tr>
<tr>
<td>1.5</td>
<td>Intrinsic Functions</td>
<td>1-8</td>
</tr>
<tr>
<td>1.6</td>
<td>Pragmas</td>
<td>1-10</td>
</tr>
<tr>
<td>1.7</td>
<td>Predefined Macros</td>
<td>1-14</td>
</tr>
</tbody>
</table>

**LIBRARIES**

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Introduction</td>
<td>2-3</td>
</tr>
<tr>
<td>2.2</td>
<td>Library Functions</td>
<td>2-4</td>
</tr>
<tr>
<td>2.2.1</td>
<td>assert.h</td>
<td>2-4</td>
</tr>
<tr>
<td>2.2.2</td>
<td>ctype.h and wctype.h</td>
<td>2-4</td>
</tr>
<tr>
<td>2.2.3</td>
<td>errno.h</td>
<td>2-5</td>
</tr>
<tr>
<td>2.2.4</td>
<td>fcntl.h</td>
<td>2-7</td>
</tr>
<tr>
<td>2.2.5</td>
<td>fenv.h</td>
<td>2-7</td>
</tr>
<tr>
<td>2.2.6</td>
<td>float.h</td>
<td>2-8</td>
</tr>
<tr>
<td>2.2.7</td>
<td>fss.h</td>
<td>2-8</td>
</tr>
<tr>
<td>2.2.8</td>
<td>inttypes.h and stdint.h</td>
<td>2-9</td>
</tr>
<tr>
<td>2.2.9</td>
<td>iso646.h</td>
<td>2-10</td>
</tr>
<tr>
<td>2.2.10</td>
<td>limits.h</td>
<td>2-10</td>
</tr>
<tr>
<td>2.2.11</td>
<td>locale.h</td>
<td>2-10</td>
</tr>
<tr>
<td>2.2.12</td>
<td>math.h and tgmath.h</td>
<td>2-11</td>
</tr>
<tr>
<td>2.2.13</td>
<td>setjmp.h</td>
<td>2-18</td>
</tr>
<tr>
<td>2.2.14</td>
<td>signal.h</td>
<td>2-18</td>
</tr>
<tr>
<td>2.2.15</td>
<td>stdarg.h</td>
<td>2-19</td>
</tr>
<tr>
<td>2.2.16</td>
<td>stdbool.h</td>
<td>2-19</td>
</tr>
<tr>
<td>2.2.17</td>
<td>stddef.h</td>
<td>2-20</td>
</tr>
<tr>
<td>2.2.18</td>
<td>stdint.h</td>
<td>2-20</td>
</tr>
<tr>
<td>2.2.19</td>
<td>stdio.h and wchar.h</td>
<td>2-20</td>
</tr>
<tr>
<td>2.2.20</td>
<td>stdlib.h and wchar.h</td>
<td>2-31</td>
</tr>
<tr>
<td>2.2.21</td>
<td>string.h and wchar.h</td>
<td>2-35</td>
</tr>
<tr>
<td>2.2.22</td>
<td>time.h and wchar.h</td>
<td>2-39</td>
</tr>
<tr>
<td>2.2.23</td>
<td>Unistd.h</td>
<td>2-42</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.2.24</td>
<td>wchar.h</td>
<td>2-43</td>
</tr>
<tr>
<td>2.2.25</td>
<td>wctype.h</td>
<td>2-44</td>
</tr>
</tbody>
</table>

**PCP ASSEMBLY LANGUAGE**

- 3.1 Introduction .................................. 3-3
- 3.2 Built-in Assembly Functions .................. 3-3
- 3.2.1 Overview of Built-in Assembly Functions .... 3-3
- 3.2.2 Detailed Description of Built-in Assembly Functions 3-6
- 3.3 Assembler Directives and Controls ............. 3-18
- 3.3.1 Overview of Assembler Directives ............ 3-18
- 3.3.2 Detailed Description of Assembler Directives 3-20
- 3.3.3 Overview of Assembler Controls .............. 3-66
- 3.3.4 Detailed Description of Assembler Controls 3-66

**RUN-TIME ENVIRONMENT**

- 4.1 Introduction .................................. 4-3
- 4.2 Startup Code .................................... 4-3
- 4.3 Stack Usage .................................... 4-3
- 4.4 Heap Allocation .................................. 4-4
- 4.5 Floating-Point Arithmetic ....................... 4-4
- 4.5.1 Compliance with IEEE-754 ...................... 4-5
- 4.5.2 Special Floating-Point Values ................. 4-6
- 4.5.3 Trapping Floating-Point Exceptions .......... 4-6
- 4.5.4 Floating-Point Trap Handling API ............ 4-8

**TOOL OPTIONS**

- 5.1 Compiler Options ............................... 5-3
- 5.2 Assembler Options .............................. 5-54
- 5.3 Linker Options .................................. 5-89
- 5.4 Control Program Options ....................... 5-130
- 5.5 Make Utility Options .......................... 5-170
- 5.6 Archiver Options ............................... 5-197
**LIST FILE FORMATS**  
6.1 Assembler List File Format ........................................... 6-3  
6.2 Linker Map File Format .................................................. 6-5  

**OBJECT FILE FORMATS**  
7.1 ELF/DWARF Object Format .............................................. 7-3  
7.2 Motorola S-Record Format ............................................... 7-4  
7.3 Intel Hex Record Format .................................................. 7-8  

**LINKER SCRIPT LANGUAGE**  
8.1 Introduction .............................................................. 8-3  
8.2 Structure of a Linker Script File ...................................... 8-3  
8.3 Syntax of the Linker Script Language ................................ 8-6  
8.3.1 Preprocessing .......................................................... 8-6  
8.3.2 Lexical Syntax .......................................................... 8-7  
8.3.3 Identifiers .............................................................. 8-7  
8.3.4 Expressions ............................................................. 8-8  
8.3.5 Built-in Functions ..................................................... 8-9  
8.3.6 LSL Definitions in the Linker Script File ......................... 8-11  
8.3.7 Memory and Bus Definitions ....................................... 8-11  
8.3.8 Architecture Definition ............................................. 8-13  
8.3.9 Derivative Definition ................................................ 8-15  
8.3.10 Processor Definition and Board Specification ................. 8-16  
8.3.11 Section Placement Definition .................................... 8-16  
8.4 Expression Evaluation ................................................... 8-21  
8.5 Semantics of the Architecture Definition ............................ 8-22  
8.5.1 Defining an Architecture .......................................... 8-23  
8.5.2 Defining Internal Buses ............................................. 8-24  
8.5.3 Defining Address Spaces ......................................... 8-24  
8.5.4 Mappings ............................................................... 8-27  
8.6 Semantics of the Derivative Definition .............................. 8-30  
8.6.1 Defining a Derivative ............................................... 8-30  
8.6.2 Instantiating Core Architectures .................................. 8-31
8.6.3 Defining Internal Memory and Buses ........................ 8-32
8.7 Semantics of the Board Specification ........................ 8-34
8.7.1 Defining a Processor ........................................ 8-34
8.7.2 Instantiating Derivatives .................................... 8-35
8.7.3 Defining External Memory and Buses ....................... 8-36
8.8 Semantics of the Section Layout Definition .................. 8-38
8.8.1 Defining a Section Layout .................................. 8-39
8.8.2 Creating and Locating Groups of Sections ................. 8-40
8.8.3 Creating or Modifying Special Sections .................... 8-46
8.8.4 Creating Symbols ............................................. 8-51
8.8.5 Conditional Group Statements ............................... 8-52

MISRA-C RULES 9-1

9.1 MISRA-C:1998 .................................................. 9-3
9.2 MISRA-C:2004 .................................................. 9-10

INDEX
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 1: PCP C Language
Contains overviews of all language extensions:
• Data types
• Keywords
• Function qualifiers
• Intrinsic functions
• Pragmas
• Predefined macros

Chapter 2: Libraries
Contains overviews of all library functions you can use in your C source. The libraries are implemented according to the ISO/IEC 9899:1999(E) standard.

Chapter 3: PCP Assembly Language
Contains an overview of all assembly functions that you can use in your assembly source code.

Chapter 4: Run-time Environment
Contains a description of the C startup code and explains stack and heap usage and floating-point arithmetic.

Chapter 5: Tool Options
Contains a description of all tool options:
• Compiler options
• Assembler options
• Linker options
• Control program options
• Make utility options
• Archiver options

Chapter 6: List File Formats
Contains a description of the following list file formats:
• Assembler List File Format
• Linker Map File Format
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

  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

  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]
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:

<table>
<thead>
<tr>
<th>Type</th>
<th>Keyword</th>
<th>Size (bit)</th>
<th>Align (bit)</th>
<th>Ranges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>_Bool</td>
<td>32</td>
<td>32</td>
<td>0 or 1</td>
</tr>
<tr>
<td>Character</td>
<td>char</td>
<td>32</td>
<td>32</td>
<td>-2^{31}..2^{31}-1</td>
</tr>
<tr>
<td></td>
<td>signed char</td>
<td>32</td>
<td>32</td>
<td>-2^{31}..2^{31}-1</td>
</tr>
<tr>
<td></td>
<td>unsigned char</td>
<td>32</td>
<td>32</td>
<td>0..2^{32}-1</td>
</tr>
<tr>
<td>Integral</td>
<td>short</td>
<td>32</td>
<td>32</td>
<td>-2^{31}..2^{31}-1</td>
</tr>
<tr>
<td></td>
<td>signed short</td>
<td>32</td>
<td>32</td>
<td>-2^{31}..2^{31}-1</td>
</tr>
<tr>
<td></td>
<td>unsigned short</td>
<td>32</td>
<td>32</td>
<td>0..2^{32}-1</td>
</tr>
<tr>
<td></td>
<td>int</td>
<td>32</td>
<td>32</td>
<td>-2^{31}..2^{31}-1</td>
</tr>
<tr>
<td></td>
<td>signed int</td>
<td>32</td>
<td>32</td>
<td>-2^{31}..2^{31}-1</td>
</tr>
<tr>
<td></td>
<td>long</td>
<td>32</td>
<td>32</td>
<td>-2^{31}..-2^{31}-1</td>
</tr>
<tr>
<td></td>
<td>unsigned long</td>
<td>32</td>
<td>32</td>
<td>0..2^{32}-1</td>
</tr>
<tr>
<td></td>
<td>long long</td>
<td>32</td>
<td>32</td>
<td>-2^{31}..-2^{31}-1</td>
</tr>
<tr>
<td></td>
<td>unsigned long long</td>
<td>32</td>
<td>32</td>
<td>0..2^{32}-1</td>
</tr>
<tr>
<td>Pointer</td>
<td>pointer to data</td>
<td>32</td>
<td>32</td>
<td>0..2^{31}-1</td>
</tr>
<tr>
<td></td>
<td>pointer to func</td>
<td>32</td>
<td>32</td>
<td>0..2^{32}-1</td>
</tr>
<tr>
<td>Floating-Point</td>
<td>float</td>
<td>32</td>
<td>32</td>
<td>-3.402e38..-1.175e-38</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>32</td>
<td>32</td>
<td>-3.402e38..-1.175e-38</td>
</tr>
<tr>
<td></td>
<td>long double</td>
<td>32</td>
<td>32</td>
<td>1.175e-38..3.402e38</td>
</tr>
</tbody>
</table>

*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.

```c
__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.
### Table 1-2: Available input/output operand constraints

<table>
<thead>
<tr>
<th>Constraint character</th>
<th>Type</th>
<th>Operand</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>Word register</td>
<td>r0 .. r7</td>
<td></td>
</tr>
<tr>
<td>number</td>
<td>Type of operand it is associated with</td>
<td>same as %number</td>
<td>Indicates that %number and number are the same register.</td>
</tr>
</tbody>
</table>

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.

```c
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)`.

```c
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.)

```c
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()`

```c
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__()`

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

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

`__free()`

```c
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()`

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

Inserts a NOP instruction.

`__get_return_address()`

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

Used by the compiler in `retjmp()`.

`__ld32_fpi()`

```c
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:

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

generates:

```c
ldl.i r5,HI(0xf0003210)
ldl.i r5,LO(0xf0003210)
ld.f r1,[r5],size=32
```

`__st32_fpi()`

```c
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:

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

generates:

```c
ldl.i r5,HI(0xf0003210)
ldl.i r5,LO(0xf0003210)
st.f r1,[r5],size=32
```

`__exit()`

```c
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 combination 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 resemblance 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`:

```c
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 optimizations should optimize 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.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BIGENDIAN</strong></td>
<td>Expands to 0.</td>
</tr>
<tr>
<td><strong>BUILD</strong></td>
<td>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 -&gt; 134000000 Build #22.1.4 -&gt; 22001004</td>
</tr>
<tr>
<td><strong>CPCP</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>CPU</strong></td>
<td>Expands to a string with the CPU supplied with the option --cpu. When no --cpu is supplied, this symbol is not defined.</td>
</tr>
<tr>
<td><strong>CORE</strong></td>
<td>Expands to a string with the core depending on the C compiler options option --cpu. The symbol expands to &quot;pcp2&quot; when the option --cpu is not specified.</td>
</tr>
<tr>
<td><strong>DATE</strong></td>
<td>Expands to the compilation date: &quot;mmm dd yyyy&quot;.</td>
</tr>
<tr>
<td><strong>DOUBLE_FP</strong></td>
<td>Expands to 0. The PCP always treats a double as float.</td>
</tr>
<tr>
<td><strong>FILE</strong></td>
<td>Expands to the current source file name.</td>
</tr>
<tr>
<td><strong>LINE</strong></td>
<td>Expands to the line number of the line where this macro is called.</td>
</tr>
<tr>
<td><strong>REVISION</strong></td>
<td>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 -&gt; 1, v1.0rb -&gt; -1</td>
</tr>
<tr>
<td>Macro</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>SFRFILE</strong>(cpu)</td>
<td>This macro expands to the filename of the used SFR file, including the &lt;&gt;. The cpu is the argument of the macro. For example, if --cpu=tc1920b is specified, the macro <strong>SFRFILE</strong>(CPU) expands to &lt;regtc1920b.sfr&gt;.</td>
</tr>
<tr>
<td><strong>SINGLE_FP</strong></td>
<td>Expands to 1. The PCP compiler always treats a double as float.</td>
</tr>
<tr>
<td><strong>STDC</strong></td>
<td>Identifies the level of ANSI standard. The macro expands to 1 if you set option --language (Control language extensions), otherwise expands to 0.</td>
</tr>
<tr>
<td><strong>STDC_HOSTED</strong></td>
<td>Always expands to 0, indicating the implementation is not a hosted implementation.</td>
</tr>
<tr>
<td><strong>STDC_VERSION</strong></td>
<td>Identifies the ISO C version number. Expands to 199901L for ISO C99 or 199409L for ISO C90.</td>
</tr>
<tr>
<td><strong>TASKING</strong></td>
<td>Identifies the compiler as a TASKING compiler. Expands to 1 if a TASKING compiler is used.</td>
</tr>
<tr>
<td><strong>TASKING_SFR</strong></td>
<td>Expands to 1 if TASKING .sfr files are used. Not defined when option --no-tasking-sfr is used.</td>
</tr>
<tr>
<td><strong>TIME</strong></td>
<td>Expands to the compilation time: &quot;hh:mm:ss&quot;</td>
</tr>
<tr>
<td><strong>VERSION</strong></td>
<td>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 -&gt; 1000, v12.3 -&gt; 12003</td>
</tr>
</tbody>
</table>

Table 1-3: Predefined macros
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.

<table>
<thead>
<tr>
<th>Library to link</th>
<th>Description</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>libc.a</td>
<td>C library</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(Some functions require the floating-point library. Also includes the startup code.)</td>
<td></td>
</tr>
<tr>
<td>libfp.a</td>
<td>Floating-point library (non-trapping)</td>
<td></td>
</tr>
<tr>
<td>libfpt.a</td>
<td>Floating-point library (trapping)</td>
<td>(Control program option --fp-trap)</td>
</tr>
</tbody>
</table>

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 implementation depends on the context where your application will run. These functions are for example all I/O related functions. Where possible, these functions are implemented using file system simulation (FSS). This system can be used by 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.

<table>
<thead>
<tr>
<th>Ctype.h</th>
<th>Wctype.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isalnum</td>
<td>iswalnum</td>
<td>Returns a non-zero value when c is an alphabetic character or a number ([A-Z][a-z][0-9]).</td>
</tr>
<tr>
<td>isalpha</td>
<td>iswalpha</td>
<td>Returns a non-zero value when c is an alphabetic character ([A-Z][a-z]).</td>
</tr>
<tr>
<td>isblank</td>
<td>iswblank</td>
<td>Returns a non-zero value when c is a blank character (tab, space...)</td>
</tr>
<tr>
<td>iscntrl</td>
<td>iswcntrl</td>
<td>Returns a non-zero value when c is a control character.</td>
</tr>
<tr>
<td>isdigit</td>
<td>iswdigit</td>
<td>Returns a non-zero value when c is a numeric character ([0-9]).</td>
</tr>
<tr>
<td>isgraph</td>
<td>iswgraph</td>
<td>Returns a non-zero value when c is printable, but not a space.</td>
</tr>
<tr>
<td>islower</td>
<td>iswlower</td>
<td>Returns a non-zero value when c is a lowercase character ([a-z]).</td>
</tr>
</tbody>
</table>
### ctype.h Description

<table>
<thead>
<tr>
<th>Ctype.h</th>
<th>Wctype.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isprint</td>
<td>iswprint</td>
<td>Returns a non-zero value when c is printable, including spaces.</td>
</tr>
<tr>
<td>ispunct</td>
<td>iswpunct</td>
<td>Returns a non-zero value when c is a punctuation character (such as '.', ',', '!', etc.).</td>
</tr>
<tr>
<td>isspace</td>
<td>iswspace</td>
<td>Returns a non-zero value when c is a space type character (space, tab, vertical tab, formfeed, linefeed, carriage return).</td>
</tr>
<tr>
<td>isupper</td>
<td>iswupper</td>
<td>Returns a non-zero value when c is an uppercase character ([A-Z]).</td>
</tr>
<tr>
<td>isxdigit</td>
<td>iswxdigit</td>
<td>Returns a non-zero value when c is a hexadecimal digit ([0-9][A-F][a-f]).</td>
</tr>
<tr>
<td>tolower</td>
<td>towlower</td>
<td>Returns c converted to a lowercase character if it is an uppercase character, otherwise c is returned.</td>
</tr>
<tr>
<td>toupper</td>
<td>towupper</td>
<td>Returns c converted to an uppercase character if it is a lowercase character, otherwise c is returned.</td>
</tr>
<tr>
<td>_tolower</td>
<td>-</td>
<td>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.</td>
</tr>
<tr>
<td>_toupper</td>
<td>-</td>
<td>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.</td>
</tr>
<tr>
<td>isascii</td>
<td></td>
<td>Returns a non-zero value when c is in the range of 0 and 127. This function is not defined in ISO/IEC 9899.</td>
</tr>
<tr>
<td>toascii</td>
<td></td>
<td>Converts c to an ASCII value (strip highest bit). This function is not defined in ISO/IEC 9899.</td>
</tr>
</tbody>
</table>

#### 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
 EPERM
 ENOENT
 EINTR
 EIO
 EBADF
 EAGAIN
 ENOMEM
 EACCES
 ENFILE
 EINVAL
 ENOTDIR
 ENFILE
 EEXIST
 ENOSPC
 EISDIR
 EINVAL
 ENFILE
 ETXTBSY
 ENOSPC
 EPIPE
 ELOOP
 ENAMETOOLONG

Floating-point errors

EDOM
 ERANGE

Errors returned by printf/scanf

ERR_FORMAT
 ERR_NOFLOAT
 ERR_NOLONG
 ERR_NOPPOINT

Error returned by file positioning routines

ERR_POS

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

EILSEQ 

No error
Not owner
No such file or directory
Interrupted system call
I/O error
Bad file number
No more processes
Not enough core
Permission denied
Bad address
File exists
Not a directory
Is a directory
Invalid argument
File table overflow
Too many open files
Text file busy
No space left on device
Illegal seek
Read-only file system
Broken pipe
Too many levels of symbolic links
File name too long
Argument too large
Result too large
Illegal format string for printf/scanf
Floating-point not supported
Long not supported
Pointers not supported
Positioning failure
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 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)

**feholdexcept**
Saves the current floating-point environment and installs an environment that ignores all floating-point exceptions. (Not implemented)

**fesetenv**
Restores a previously saved (fegetenv or feholdexcept) floating-point environment. (Not implemented)

**feupdateenv**
Saves the currently raised floating-point exceptions, restores a previously saved floating-point environment and finally raises the saved exceptions.
(Not implemented)

**feclearexcept**
Clears the current exception status flags corresponding to the flags specified in the argument. (Not implemented)

**fegetexceptflag**
Stores the current setting of the floating-point status flags. (Not implemented)

**feraiseexcept**
Raises the exceptions represented in the argument. As a result, other exceptions may be raised as well.
(Not implemented)

**fesetexceptflag**
Sets the current floating-point status flags.
(Not implemented)

**fetestexcept**
Returns the bitwise-OR of the exception macros corresponding to the exception flags which are currently set and are specified in the argument. (Not implemented)

For each supported exception, a macro is defined. The following exceptions are defined:
FE_DIVBYZERO FE_INEXACT FE_INVALID
FE_OVERFLOW FE_UNDERFLOW FE_ALL_EXCEPT

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

fesetround
Sets the current rounding directions. (Not implemented)

Currently no rounding mode macros are implemented.

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), isninf(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.

<table>
<thead>
<tr>
<th>Stdio.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_fss_break(void)</td>
<td>Buffer and breakpoint functions for CrossView Pro.</td>
</tr>
<tr>
<td>_fss_init(fd,is_close)</td>
<td>Opens file descriptors 0 (stdin), 1 (stdout) and 2 (stderr) and associates them with terminal window FSS 0 of CrossView Pro.</td>
</tr>
<tr>
<td>_close(fd)</td>
<td>See Low Level File Access Functions in section 2.2.19, Stdio.h.</td>
</tr>
<tr>
<td>_lseek(fd,offset,whence)</td>
<td></td>
</tr>
<tr>
<td>_open(fd,flags)</td>
<td></td>
</tr>
<tr>
<td>_read(fd,*buff,cnt)</td>
<td></td>
</tr>
<tr>
<td>_unlink(*name)</td>
<td></td>
</tr>
<tr>
<td>_write(fd,*buffer,cnt)</td>
<td></td>
</tr>
</tbody>
</table>
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 includes stdint.h and adds portable formatting and conversion functions. Below the conversion functions from inttypes.h are listed.

```c
intmax_t imaxabs(intmax_t j);
imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);

intmax_t strtoimax(const char * restrict nptr, char ** restrict endptrl, int base);
uintmax_t strtoumax(const char * restrict nptr, char ** restrict endptrl, int base);
intmax_t wcstoiimax(const wchar_t * restrict nptr, wchar_t ** restrict endptrl, int base);
uintmax_t wcstoumax(const wchar_t * restrict nptr, wchar_t ** restrict endptrl, int base);
```

- `imaxabs(j)`: Returns the absolute value of `j`.
- `imaxdiv(numer, denom)`: Computes `numer/denom` and `numer % denom`. The result is stored in the `quot` and `rem` components of the `imaxdiv_t` structure type.
- `strtoimax(nptr, base)`: Convert string to maximum sized integer. (Compare `strtoll`)
- `strtoumax(nptr, base)`: Convert string to maximum sized unsigned integer. (Compare `strtoull`)
- `wcstoiimax(nptr, base)`: Convert wide string to maximum sized integer. (Compare `wcstoll`)
- `wcstoumax(nptr, base)`: Convert wide string to maximum sized unsigned integer. (Compare `wcstoul`
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 ldconv *localeconv( void )

Returns a pointer to type struct ldconv with values appropriate for the formatting of numeric quantities according to the rules of the current locale. The struct ldconv 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, functionf, functionl. 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

<table>
<thead>
<tr>
<th>Math.h</th>
<th>Tgmath.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sin</td>
<td>sin</td>
<td>Returns the sine of x.</td>
</tr>
<tr>
<td>cosf</td>
<td>cos</td>
<td>Returns the cosine of x.</td>
</tr>
<tr>
<td>tanf</td>
<td>tan</td>
<td>Returns the tangent of x.</td>
</tr>
<tr>
<td>asinf</td>
<td>asin</td>
<td>Returns the arc sine ( \sin^{-1}(x) ) of x.</td>
</tr>
<tr>
<td>acosf</td>
<td>acos</td>
<td>Returns the arc cosine ( \cos^{-1}(x) ) of x.</td>
</tr>
<tr>
<td>atanf</td>
<td>atan</td>
<td>Returns the arc tangent ( \tan^{-1}(x) ) of x.</td>
</tr>
<tr>
<td>atan2f</td>
<td>atan2</td>
<td>Returns the result of: ( \tan^{-1}(y/x) ).</td>
</tr>
<tr>
<td>sinh</td>
<td>sinh</td>
<td>Returns the hyperbolic sine of x.</td>
</tr>
<tr>
<td>coshf</td>
<td>cosh</td>
<td>Returns the hyperbolic cosine of x.</td>
</tr>
<tr>
<td>tanhf</td>
<td>tanh</td>
<td>Returns the hyperbolic tangent of x.</td>
</tr>
<tr>
<td>asinh</td>
<td>asinh</td>
<td>Returns the arc hyperbolic sinus of x.</td>
</tr>
</tbody>
</table>
### Math.h | Tgmath.h | Description
---|---|---
acosh | acosh | Returns the non-negative arc hyperbolic cosine of x.
atanh | 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 | exp | Returns the result of the exponential function $e^x$.
exp2 | exp2 | Returns the result of the exponential function $2^x$. *(Not implemented)*
expm1 | expm1 | Returns the result of the exponential function $e^x-1$. *(Not implemented)*
log | log | Returns the natural logarithm $\ln(x)$, $x>0$.
log10 | log10 | Returns the base-10 logarithm of $x$, $x>0$.
log1p | log1p | Returns the base-e logarithm of $(1+x)$, $x <> -1$. *(Not implemented)*
log2 | log2 | Returns the base-2 logarithm of $x$. $x>0$. *(Not implemented)*
ilogb | ilogbf | ilogbl | i logb | Returns the signed exponent of x as an integer, $x>0$. *(Not implemented)*
logbf | logbf | logbl | logb | Returns the exponent of $x$ as a signed integer in value in floating-point notation, $x > 0$. *(Not implemented)*
### Rounding functions

<table>
<thead>
<tr>
<th>Math.h</th>
<th>Tgmath.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ceil</td>
<td>ceilf</td>
<td>ceil</td>
</tr>
<tr>
<td>floor</td>
<td>floorf</td>
<td>floor</td>
</tr>
<tr>
<td>rint</td>
<td>rintf</td>
<td>rintl</td>
</tr>
<tr>
<td>lrint</td>
<td>lrintf</td>
<td>lrintl</td>
</tr>
<tr>
<td>llrint</td>
<td>llrintf</td>
<td>llrintl</td>
</tr>
<tr>
<td>nearbyint</td>
<td>nearbyintf</td>
<td>nearbyintl</td>
</tr>
<tr>
<td>round</td>
<td>roundf</td>
<td>roundl</td>
</tr>
<tr>
<td>lround</td>
<td>lroundf</td>
<td>lroundl</td>
</tr>
<tr>
<td>llround</td>
<td>llroundf</td>
<td>llroundl</td>
</tr>
<tr>
<td>trunc</td>
<td>truncf</td>
<td>truncl</td>
</tr>
</tbody>
</table>
**Remainder after division**

<table>
<thead>
<tr>
<th>Math.h</th>
<th>Tgmath.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmod</td>
<td>fmodf</td>
<td>Returns the remainder $r$ of $x-ny$. $n$ is chosen as $\text{trunc}(x/y)$. $r$ has the same sign as $x$.</td>
</tr>
<tr>
<td>remainder</td>
<td>remainderf</td>
<td>Returns the remainder $r$ of $x-ny$. $n$ is chosen as $\text{trunc}(x/y)$. $r$ may not have the same sign as $x$. <em>(Not implemented)</em></td>
</tr>
<tr>
<td>remainderl</td>
<td>remquo</td>
<td>Same as remainder. In addition, the argument $*\text{quo}$ is given a specific value (see ISO). <em>(Not implemented)</em></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Math.h</th>
<th>Tgmath.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>frexp</td>
<td>frexf</td>
<td>Splits a float $x$ into fraction $f$ and exponent $n$, so that: $f = 0.0$ or $0.5 \leq</td>
</tr>
<tr>
<td>ldexp</td>
<td>ldexpf</td>
<td>Inverse of frexp. Returns the result of $x*2^n$. ($x$ and $n$ are both arguments).</td>
</tr>
<tr>
<td>modf</td>
<td>modff</td>
<td>Splits a float $x$ into fraction $f$ and integer $n$, so that: $</td>
</tr>
<tr>
<td>scalbn</td>
<td>scalbnf</td>
<td>Computes the result of $x*\text{FLT_RADIX}^n$, efficiently, not normally by computing FLT_RADIX$^n$ explicitly.</td>
</tr>
<tr>
<td>scalbln</td>
<td>scalblnlf</td>
<td>Same as scalbn but with argument $n$ as long int.</td>
</tr>
</tbody>
</table>
### Power and absolute-value functions

<table>
<thead>
<tr>
<th>Math.h</th>
<th>Tgmath.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cbrt</td>
<td>cbrt</td>
<td>Returns the real cube root of (x) ((=x^{1/3})). <em>(Not implemented)</em></td>
</tr>
<tr>
<td>fabs</td>
<td>fabs</td>
<td>Returns the absolute value of (</td>
</tr>
<tr>
<td>fma</td>
<td>fma</td>
<td>Floating-point multiply add. Returns (x*y+z). <em>(Not implemented)</em></td>
</tr>
<tr>
<td>hypot</td>
<td>hypot</td>
<td>Returns the square root of (x^2+y^2).</td>
</tr>
<tr>
<td>pow</td>
<td>power</td>
<td>Returns (x) raised to the power (y) ((xy)).</td>
</tr>
<tr>
<td>sqrt</td>
<td>sqrt</td>
<td>Returns the non-negative square root of (x). (x \neq 0).</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Math.h</th>
<th>Tgmath.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>copysign</td>
<td>copysign</td>
<td>Returns the value of (x) with the sign of (y).</td>
</tr>
<tr>
<td>nan</td>
<td>nan</td>
<td>Returns a quiet NaN, if available, with content indicated through tagp. <em>(Not implemented)</em></td>
</tr>
<tr>
<td>nextafter</td>
<td>nextafter</td>
<td>Returns the next representable value in the specified format after (x) in the direction of (y). Returns (y) if (x=y). <em>(Not implemented)</em></td>
</tr>
<tr>
<td>nexttoward</td>
<td>nexttoward</td>
<td>Same as nextafter, except that the second argument in all three variants is of type long double. Returns (y) if (x=y). <em>(Not implemented)</em></td>
</tr>
</tbody>
</table>
### Positive difference, maximum, minimum

<table>
<thead>
<tr>
<th>Math.h</th>
<th>Tgmath.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdim</td>
<td>fdimf</td>
<td>Returns the positive difference between: (</td>
</tr>
<tr>
<td></td>
<td>fdiml</td>
<td>[(Not\ implemented)]</td>
</tr>
<tr>
<td>fmax</td>
<td>fmaxf</td>
<td>Returns the maximum value of their arguments.</td>
</tr>
<tr>
<td></td>
<td>fmaxl</td>
<td>[(Not\ implemented)]</td>
</tr>
<tr>
<td>fmin</td>
<td>fminf</td>
<td>Returns the minimum value of their arguments.</td>
</tr>
<tr>
<td></td>
<td>fminl</td>
<td>[(Not\ implemented)]</td>
</tr>
</tbody>
</table>

### Error and gamma (Not implemented)

<table>
<thead>
<tr>
<th>Math.h</th>
<th>Tgmath.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>erf</td>
<td>erff</td>
<td>Computes the error function of (x). [(Not\ implemented)]</td>
</tr>
<tr>
<td></td>
<td>erfl</td>
<td></td>
</tr>
<tr>
<td>erfc</td>
<td>erfcf</td>
<td>Computes the complementary error function of (x). [(Not\ implemented)]</td>
</tr>
<tr>
<td></td>
<td>erfcl</td>
<td></td>
</tr>
<tr>
<td>lgamma</td>
<td>lgammaf</td>
<td>Computes the (\log_e</td>
</tr>
<tr>
<td></td>
<td>lgammal</td>
<td></td>
</tr>
<tr>
<td>tgamma</td>
<td>tgammaf</td>
<td>Computes (\Gamma(x)) [(Not\ implemented)]</td>
</tr>
<tr>
<td></td>
<td>tgammal</td>
<td></td>
</tr>
</tbody>
</table>
**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 therefore do not have a parallel function in `tgmath.h`. All arguments must be expressions of real-floating type.

<table>
<thead>
<tr>
<th>Math.h</th>
<th>Tgmath.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isgreater</td>
<td>-</td>
<td>Returns the value of ((x) &gt; (y))</td>
</tr>
<tr>
<td>isgreaterequal</td>
<td>-</td>
<td>Returns the value of ((x) &gt;= (y))</td>
</tr>
<tr>
<td>isless</td>
<td>-</td>
<td>Returns the value of ((x) &lt; (y))</td>
</tr>
<tr>
<td>islessequal</td>
<td>-</td>
<td>Returns the value of ((x) &lt;= (y))</td>
</tr>
<tr>
<td>islessgreater</td>
<td>-</td>
<td>Returns the value of ((x) &lt; (y) | | (x) &gt; (y))</td>
</tr>
<tr>
<td>isunordered</td>
<td>-</td>
<td>Returns 1 if its arguments are unordered, 0 otherwise.</td>
</tr>
</tbody>
</table>

**Classification macros**

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

<table>
<thead>
<tr>
<th>Math.h</th>
<th>Tgmath.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fpclassify</td>
<td>-</td>
<td>Returns the class of its argument: (FP_INFINITE, FP_NAN, FP_NORMAL, FP_SUBNORMAL or FP_ZERO)</td>
</tr>
<tr>
<td>isfinite</td>
<td>-</td>
<td>Returns a nonzero value if and only if its argument has a finite value</td>
</tr>
<tr>
<td>isnf</td>
<td>-</td>
<td>Returns a nonzero value if and only if its argument has an infinite value</td>
</tr>
<tr>
<td>isnan</td>
<td>-</td>
<td>Returns a nonzero value if and only if its argument has NaN value.</td>
</tr>
<tr>
<td>isnormal</td>
<td>-</td>
<td>Returns a nonzero value if and only if its argument has a normal value.</td>
</tr>
<tr>
<td>signbit</td>
<td>-</td>
<td>Returns a nonzero value if and only if its argument value is negative.</td>
</tr>
</tbody>
</table>
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 erroneous arithmetic operation (for example, zero divide, overflow)
- `SIGSEGV` 4  An invalid access to storage
- `SIGTERM` 5  A termination request sent to the program
- `SIGABRT` 6  Abnormal termination, such as is initiated by the `abort` function

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

- `int raise(int sig)`

The next function determines how subsequent signals will be handled:

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

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

- `SIG_DFL` Default behavior is used
- `SIG_IGN` The signal is ignored

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

```c
#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:

ptdiff_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

<table>
<thead>
<tr>
<th>Stdio.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUFSIZ  512</td>
<td>Size of the buffer used by the <code>setbuf/setvbuf</code> function: 512</td>
</tr>
<tr>
<td>EOF     -1</td>
<td>End of file indicator.</td>
</tr>
</tbody>
</table>
| 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 |
| _IOPBF | Expand to an integer expression, suitable for use as argument to the `setvbuf` function. |
| _IOLBF |  |
| _IONBF |  |
| L_tmpnam | Size of the string used to hold temporary file names: 8 (tmpxxxxxx) |
| TMP_MAX 0x8000 | Maximum number of unique temporary filenames that can be generated: 0x8000 |
| stderr | Expressions of type "pointer to FILE" that point to the FILE objects associated with standard error, input and output streams. |
| stdin |  |
| stdout |  |
Low level file access functions

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

File access

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

**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 built in order:

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

- A number specifying a minimum field width. The converted argument is printed in a field with at least the length specified here. If the converted argument has fewer characters than specified, it will be padded at the left side (or at the right when the flag ‘-’ was specified) with spaces. Padding to numeric fields will be done with zeros when the flag ‘0’ is also specified (only when padding left). Instead of a numeric value, also ‘*’ may be specified, the value is then taken from the next argument, which is assumed to be of type int.

- A period. This separates the minimum field width from the precision.

- A number specifying the maximum length of a string to be printed. Or the number of digits printed after the decimal point (only for floating-point conversions). Or the minimum number of digits to be printed for an integer conversion. Instead of a numeric value, also ‘*’ may be specified, the value is then taken from the next argument, which is assumed to be of type int.

- A length modifier 'h', '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:

<table>
<thead>
<tr>
<th>Character</th>
<th>Printed as</th>
</tr>
</thead>
<tbody>
<tr>
<td>d, i</td>
<td>int, signed decimal</td>
</tr>
<tr>
<td>o</td>
<td>int, unsigned octal</td>
</tr>
<tr>
<td>x, X</td>
<td>int, unsigned hexadecimal in lowercase or uppercase respectively</td>
</tr>
<tr>
<td>u</td>
<td>int, unsigned decimal</td>
</tr>
<tr>
<td>c</td>
<td>int, single character (converted to unsigned char)</td>
</tr>
<tr>
<td>Character</td>
<td>Printed as</td>
</tr>
<tr>
<td>-----------</td>
<td>------------</td>
</tr>
<tr>
<td>s</td>
<td>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</td>
</tr>
<tr>
<td>f</td>
<td>double</td>
</tr>
<tr>
<td>e, E</td>
<td>double</td>
</tr>
<tr>
<td>g, G</td>
<td>double</td>
</tr>
<tr>
<td>n</td>
<td>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.</td>
</tr>
<tr>
<td>p</td>
<td>pointer (hexadecimal 24-bit value)</td>
</tr>
<tr>
<td>%</td>
<td>No argument is converted, a ‘%’ is printed.</td>
</tr>
</tbody>
</table>

*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.

<table>
<thead>
<tr>
<th>Character</th>
<th>Scanned as</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>int, signed decimal.</td>
</tr>
<tr>
<td>i</td>
<td>int, the integer may be given octal (i.e. a leading 0 is entered) or hexadecimal (leading &quot;0x&quot; or &quot;0X&quot;), or just decimal.</td>
</tr>
<tr>
<td>o</td>
<td>int, unsigned octal.</td>
</tr>
<tr>
<td>u</td>
<td>int, unsigned decimal.</td>
</tr>
<tr>
<td>x</td>
<td>int, unsigned hexadecimal in lowercase or uppercase.</td>
</tr>
<tr>
<td>c</td>
<td>single character (converted to unsigned char).</td>
</tr>
<tr>
<td>s</td>
<td>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.</td>
</tr>
<tr>
<td>f</td>
<td>float</td>
</tr>
<tr>
<td>e, E</td>
<td>float</td>
</tr>
<tr>
<td>g, G</td>
<td>float</td>
</tr>
<tr>
<td>n</td>
<td>int *, the number of characters written so far is written into the argument. No scanning is done.</td>
</tr>
<tr>
<td>p</td>
<td>pointer; hexadecimal 24-bit value which must be entered without 0x– prefix.</td>
</tr>
<tr>
<td>[...]</td>
<td>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.</td>
</tr>
<tr>
<td>[^...]</td>
<td>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.</td>
</tr>
<tr>
<td>%</td>
<td>Literal '%', no assignment is done.</td>
</tr>
</tbody>
</table>

*Table 2-3: Scanf conversion characters*
<table>
<thead>
<tr>
<th>Stdio.h</th>
<th>Wchar.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fgetc(stream)</code></td>
<td><code>fgetwc(stream)</code></td>
<td>Reads one character from <code>stream</code>. Returns the read character, or EOF/WEOF on error. <em>(FSS implementation)</em></td>
</tr>
<tr>
<td><code>getc(stream)</code></td>
<td><code>getwc(stream)</code></td>
<td>Same as <code>fgetc/fgetwc</code> except that is implemented as a macro. <em>(FSS implementation)</em></td>
</tr>
<tr>
<td><code>getchar(stdin)</code></td>
<td><code>getwchar(stdin)</code></td>
<td>Reads one character from the <code>stdin</code> stream. Returns the character read or EOF/WEOF on error. Implemented as macro. <em>(FSS implementation)</em></td>
</tr>
<tr>
<td><code>fgets(*s, n, stream)</code></td>
<td><code>fgetws(*s, n, stream)</code></td>
<td>Reads at most the next <code>n-1</code> characters from the <code>stream</code> into array <code>s</code> until a newline is found. Returns <code>s</code> or NULL or EOF/WEOF on error. <em>(FSS implementation)</em></td>
</tr>
<tr>
<td><code>gets(*s, n, stdin)</code></td>
<td></td>
<td>Reads at most the next <code>n-1</code> characters from the <code>stdin</code> stream into array <code>s</code>. A newline is ignored. Returns <code>s</code> or NULL or EOF/WEOF on error. <em>(FSS implementation)</em></td>
</tr>
<tr>
<td><code>ungetc(c, stream)</code></td>
<td><code>ungetwc(c, stream)</code></td>
<td>Pushes character <code>c</code> back onto the input <code>stream</code>. Returns EOF/WEOF on error.</td>
</tr>
<tr>
<td><code>fscanf(stream, format,...)</code></td>
<td><code>fwscanf(stream, format,...)</code></td>
<td>Performs a formatted read from the given <code>stream</code>. Returns the number of items converted successfully. <em>(FSS implementation)</em></td>
</tr>
<tr>
<td>Stdio.h</td>
<td>Wchar.h</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>scanf(format,...)</code></td>
<td><code>wscanf(format,...)</code></td>
<td>Performs a formatted read from stdin. Returns the number of items converted successfully. <em>(FSS implementation)</em></td>
</tr>
<tr>
<td><code>sscanf(*s, format,...)</code></td>
<td><code>swscanf(*s, format,...)</code></td>
<td>Performs a formatted read from the string s. Returns the number of items converted successfully.</td>
</tr>
<tr>
<td><code>vscanf(format, arg)</code></td>
<td><code>vswscanf(format, arg)</code></td>
<td>Same as <code>scanf/wscanf</code>, but extra arguments are given as variable argument list <code>arg</code>. <em>(See section 2.2.15, stdarg.h)</em></td>
</tr>
<tr>
<td><code>vfscanf(stream, format, arg)</code></td>
<td><code>vfwscanf(stream, format, arg)</code></td>
<td>Same as <code>fscanf/fwscanf</code>, but extra arguments are given as variable argument list <code>arg</code>. <em>(See section 2.2.15, stdarg.h)</em></td>
</tr>
<tr>
<td><code>putc(c, stream)</code></td>
<td><code>putwc(c, stream)</code></td>
<td>Put character c onto the given stream. Returns EOF/WEOF on error. <em>(FSS implementation)</em></td>
</tr>
<tr>
<td><code>putchar(c, stdout)</code></td>
<td><code>putwchar(c, stdout)</code></td>
<td>Put character c onto the stdout stream. Returns EOF/WEOF on error. Implemented as macro. <em>(FSS implementation)</em></td>
</tr>
<tr>
<td><code>fgets(*s, stream)</code></td>
<td><code>fputws(*s, stream)</code></td>
<td>Writes string s to the given stream. Returns EOF/WEOF on error. <em>(FSS implementation)</em></td>
</tr>
<tr>
<td>Library</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td><strong>Stdio.h</strong></td>
<td><strong>Wchar.h</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><code>puts(*s)</code></td>
<td>-</td>
<td>Writes string <code>s</code> to the stdout stream. Returns EOF/WEOF on error. <em>(FSS implementation)</em></td>
</tr>
<tr>
<td><code>fprintf(stream, format,...)</code></td>
<td><code>fwprintf(stream, format,...)</code></td>
<td>Performs a formatted write to the given <code>stream</code>. Returns EOF/WEOF on error. <em>(FSS implementation)</em></td>
</tr>
<tr>
<td><code>printf(format, ...)</code></td>
<td><code>wprintf(format, ...)</code></td>
<td>Performs a formatted write to the stream stdout. Returns EOF/WEOF on error. <em>(FSS implementation)</em></td>
</tr>
<tr>
<td><code>sprintf(*s, format,...)</code></td>
<td>-</td>
<td>Performs a formatted write to string <code>s</code>. Returns EOF/WEOF on error.</td>
</tr>
<tr>
<td><code>snprintf(*s, n, format,...)</code></td>
<td><code>swprintf(*s, n, format,...)</code></td>
<td>Same as <code>sprintf</code>, but <code>n</code> specifies the maximum number of characters (including the terminating null character) to be written.</td>
</tr>
<tr>
<td><code>vfprintf(stream, format, arg)</code></td>
<td><code>vfwprintf(stream, format, arg)</code></td>
<td>Same as <code>fprintf/fwprintf</code>, but extra arguments are given as variable argument list <code>arg</code>. *(See section 2.2.15, <code>stdarg.h</code>) <em>(FSS implementation)</em></td>
</tr>
<tr>
<td><code>vprintf(format, arg)</code></td>
<td><code>vwprintf(format, arg)</code></td>
<td>Same as <code>printf/wprintf</code>, but extra arguments are given as variable argument list <code>arg</code>. *(See section 2.2.15, <code>stdarg.h</code>) <em>(FSS implementation)</em></td>
</tr>
<tr>
<td><code>vsprintf(*s, format, arg)</code></td>
<td><code>vswprintf(*s, format, arg)</code></td>
<td>Same as <code>sprintf/swprintf</code>, but extra arguments are given as variable argument list <code>arg</code>. <em>(See section 2.2.15, <code>stdarg.h</code>)</em></td>
</tr>
</tbody>
</table>
Direct input/output

<table>
<thead>
<tr>
<th>Stdio.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fread(ptr, size, nobj, stream)</code></td>
<td>Reads <code>nobj</code> members of <code>size</code> bytes from the given <code>stream</code> into the array pointed to by <code>ptr</code>. Returns the number of elements successfully read. <code>(FSS implementation)</code></td>
</tr>
<tr>
<td><code>fwrite((ptr, size, nobj, stream)</code></td>
<td>Writes <code>nobj</code> members of <code>size</code> bytes from to the array pointed to by <code>ptr</code> to the given <code>stream</code>. Returns the number of elements successfully written. <code>(FSS implementation)</code></td>
</tr>
</tbody>
</table>

Random access

<table>
<thead>
<tr>
<th>Stdio.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fseek(stream, offset, origin)</code></td>
<td>Sets the position indicator for <code>stream</code>. <code>(FSS implementation)</code></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ftell(stream)</code></td>
<td>Returns the current file position for <code>stream</code>, or -1L on error. <code>(FSS implementation)</code></td>
</tr>
<tr>
<td><code>rewind(stream)</code></td>
<td>Sets the file position indicator for the <code>stream</code> to the beginning of the file. This function is equivalent to: (void) fseek(stream,0L,SEEK_SET); clearerr(stream); <code>(FSS implementation)</code></td>
</tr>
<tr>
<td><code>fgetpos(stream,pos)</code></td>
<td>Stores the current value of the file position indicator for <code>stream</code> in the object pointed to by <code>pos</code>. <code>(FSS implementation)</code></td>
</tr>
<tr>
<td><code>fsetpos(stream,pos)</code></td>
<td>Positions <code>stream</code> at the position recorded by <code>fgetpos</code> in <code>*pos</code>. <code>(FSS implementation)</code></td>
</tr>
</tbody>
</table>
### Operations on files

<table>
<thead>
<tr>
<th>Stdio.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>remove(file)</td>
<td>Removes the named file, so that a subsequent attempt to open it fails. Returns a non-zero value if not successful.</td>
</tr>
<tr>
<td>rename(old,new)</td>
<td>Changes the name of the file from old name to new name. Returns a non-zero value if not successful.</td>
</tr>
<tr>
<td>tmpfile()</td>
<td>Creates a temporary file of the mode &quot;wb+&quot; that will be automatically removed when closed or when the program terminates normally. Returns a file pointer.</td>
</tr>
<tr>
<td>tmpnam(buffer)</td>
<td>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 &quot;tmp&quot;. At most TMP_MAX unique file names can be generated.</td>
</tr>
</tbody>
</table>

### Error handling

<table>
<thead>
<tr>
<th>Stdio.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clearerr(stream)</td>
<td>Clears the end of file and error indicators for stream.</td>
</tr>
<tr>
<td>ferror(stream)</td>
<td>Returns a non-zero value if the error indicator for stream is set.</td>
</tr>
<tr>
<td>feof(stream)</td>
<td>Returns a non-zero value if the end of file indicator for stream is set.</td>
</tr>
<tr>
<td>perror(*s)</td>
<td>Prints s and the error message belonging to the integer errno. (See section 2.2.3, errno.h)</td>
</tr>
</tbody>
</table>

#### 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
- Environment 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_SUCCESS** 0: Predefined exit codes that can be used in the `exit` function.
- **EXIT_FAILURE** 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 initial portion of a string *s* to a `double`, `int`, `long int` and `long long int` value respectively.

```c
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.

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>long strtol (*s,**endp,base)</td>
<td>Converts string to a long</td>
</tr>
<tr>
<td>long long strtoll (*s,**endp,base)</td>
<td>Converts string to a long long</td>
</tr>
<tr>
<td>unsigned long strtoul (*s,**endp,base)</td>
<td>Converts string to an unsigned long</td>
</tr>
<tr>
<td>unsigned long long strtoull (*s,**endp,base)</td>
<td>Converts string to an unsigned long long</td>
</tr>
</tbody>
</table>

**Wchar.h**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>long wcstol (*s,**endp,base)</td>
<td>Converts string to a long</td>
</tr>
<tr>
<td>long long wcstoll (*s,**endp,base)</td>
<td>Converts string to a long long</td>
</tr>
<tr>
<td>unsigned long wcstoul (*s,**endp,base)</td>
<td>Converts string to an unsigned long</td>
</tr>
<tr>
<td>unsigned long long wcstoull (*s,**endp,base)</td>
<td>Converts string to an unsigned long long</td>
</tr>
</tbody>
</table>

**Random number generation**

- **rand**
  - Returns a pseudo random integer in the range 0 to RAND_MAX.

- **srand(seed)**
  - Same as rand but uses *seed* for a new sequence of pseudo random numbers.

**Memory management**

- **malloc(size)**
  - Allocates space for an object with size *size*. The allocated space is not initialized. Returns a pointer to the allocated space.

- **calloc(nobj,size)**
  - Allocates space for n objects with size *size*. The allocated space is initialized with zeros. Returns a pointer to the allocated space.

- **free(*ptr)**
  - Deallocates the memory space pointed to by *ptr* which should be a pointer earlier returned by the malloc or calloc function.

- **realloc(*ptr,size)**
  - Deallocates the old object pointed to by *ptr* and returns a pointer to a new object with size *size*. The new object cannot have a size larger than the previous object.
Environment communication

abort()  Causes abnormal program termination. If the signal SIGABRT is 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_SUCCESS or EXIT_FAILURE.

_exit(status)  Same as exit, but not registered by the atexit function or signal handlers registered by the signal function are called.

getenv(*s)  Searches an environment list for a string s. Returns a pointer to the contents of s. NOTE: this function is not implemented because there is no OS.

system(*s)  Passes the string s to the environment for execution. NOTE: this function is not implemented because there is no OS.

Searching and sorting

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

qsort(*base,n, size,*cmp)  This function sorts an array of n members using the quick sort algorithm. The initial base of the array is given by base. The size of each member is specified by size. The array is sorted in ascending order, according to the results of the function pointed to by cmp.
**Integer arithmetic**

```
int abs(j)  Compute the absolute value of an int, long
long labs(j) int, and long long int/ respectively.
long long llabs(j)
```

```
div_t div(x,y)  Compute x/y and x%y in a single operation. X
ldiv_t ldiv(x,y) and y have respectively type int, long int and
lldiv_t lldiv(x,y) 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
```

```
```

mbstowcs 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 wcstowmb 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

<table>
<thead>
<tr>
<th>Stdio.h</th>
<th>Wchar.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>memcpy(*s1,*s2,n)</code></td>
<td><code>wmemcpy(*s1,*s2,n)</code></td>
<td>Copies $n$ characters from *s2 into *s1 and returns *s1. If *s1 and *s2 overlap the result is undefined.</td>
</tr>
<tr>
<td><code>memmove(*s1,*s2,n)</code></td>
<td><code>wmemmove(*s1,*s2,n)</code></td>
<td>Same as <code>memcpy</code>, but overlapping strings are handled correctly. Returns *s1.</td>
</tr>
<tr>
<td><code>strcpy(*s1,*s2)</code></td>
<td><code>wcscpy(*s1,*s2)</code></td>
<td>Copies *s2 into *s1 and returns *s1. If *s1 and *s2 overlap the result is undefined.</td>
</tr>
<tr>
<td><code>strncpy(*s1,*s2,n)</code></td>
<td><code>wcsncpy(*s1,*s2,n)</code></td>
<td>Copies not more than $n$ characters from *s2 into *s1 and returns *s1. If *s1 and *s2 overlap the result is undefined.</td>
</tr>
<tr>
<td><code>strcat(*s1,*s2)</code></td>
<td><code>wcscat(*s1,*s2)</code></td>
<td>Appends a copy of *s2 to *s1 and returns *s1. If *s1 and *s2 overlap the result is undefined.</td>
</tr>
<tr>
<td><code>strncat(*s1,*s2,n)</code></td>
<td><code>wcsncat(*s1,*s2,n)</code></td>
<td>Appends not more than $n$ characters from *s2 to *s1 and returns *s1. If *s1 and *s2 overlap the result is undefined.</td>
</tr>
</tbody>
</table>
### Comparison functions

<table>
<thead>
<tr>
<th>Stdio.h</th>
<th>Wchar.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>memcmp(*s1, *s2, n)</code></td>
<td><code>wmemcmp(*s1, *s2, n)</code></td>
<td>Compares the first ( n ) characters of ( *s1 ) to the first ( n ) characters of ( *s2 ). Returns &lt; 0 if ( *s1 &lt; *s2 ), 0 if ( *s1 = = *s2 ), or &gt; 0 if ( *s1 &gt; *s2 ).</td>
</tr>
<tr>
<td><code>strcmp(*s1,*s2)</code></td>
<td><code>wcscmp(*s1,*s2)</code></td>
<td>Compares string ( *s1 ) to ( *s2 ). Returns &lt; 0 if ( *s1 &lt; *s2 ), 0 if ( *s1 = = *s2 ), or &gt; 0 if ( *s1 &gt; *s2 ).</td>
</tr>
<tr>
<td><code>strncmp(*s1,*s2,n)</code></td>
<td><code>wcsncmp(*s1,*s2,n)</code></td>
<td>Compares the first ( n ) characters of ( *s1 ) to the first ( n ) characters of ( *s2 ). Returns &lt; 0 if ( *s1 &lt; *s2 ), 0 if ( *s1 = = *s2 ), or &gt; 0 if ( *s1 &gt; *s2 ).</td>
</tr>
<tr>
<td><code>strcoll(*s1,*s2)</code></td>
<td><code>wcscoll(*s1,*s2)</code></td>
<td>Performs a local–specific comparison between string ( *s1 ) and string ( *s2 ) according to the LC_COLLATE category of the current locale. Returns &lt; 0 if ( *s1 &lt; *s2 ), 0 if ( *s1 = = *s2 ), or &gt; 0 if ( *s1 &gt; *s2 ). (See section 2.2.11, <code>locale.h</code>)</td>
</tr>
<tr>
<td><code>strxfrm(*s1,*s2,n)</code></td>
<td><code>wcsxfrm(*s1,*s2,n)</code></td>
<td>Transforms (a local) string ( *s2 ) so that a comparison between transformed strings with <code>strcmp</code> gives the same result as a comparison between non–transformed strings with <code>strcoll</code>. Returns the transformed string ( *s1 ).</td>
</tr>
</tbody>
</table>
### Search functions

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

<table>
<thead>
<tr>
<th>Stdio.h</th>
<th>Wchar.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>memset(*s,c,n)</td>
<td>wmemset(*s,c,n)</td>
<td>Fills the first n bytes of *s with character c and returns *s.</td>
</tr>
<tr>
<td>strerror(errno)</td>
<td>-</td>
<td>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)</td>
</tr>
<tr>
<td>strlen(*s)</td>
<td>wcslen(*s)</td>
<td>Returns the length of string *s.</td>
</tr>
</tbody>
</table>

### 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 defined as follows:

```c
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 \( \text{time}_t \). The return value has the same encoding as the return value of the \( \text{time} \) 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 calendar time pointed to by \( timer \) to local time in the form of a string. This is equivalent to: \( \text{asctime}(\text{localtime}(\text{timer})) \)

**gmtime(*timer)**
Converts the calendar 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`:

<table>
<thead>
<tr>
<th>Stdio.h</th>
<th>Wchar.h</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>strftime(*s,smax,*fmt,tm *tp)</code></td>
<td><code>wstrftime(*s,smax,*fmt,tm *tp)</code></td>
</tr>
</tbody>
</table>

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:

```c
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 characters is 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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Equivalent to locale specific test</th>
</tr>
</thead>
<tbody>
<tr>
<td>iswalnum(wc)</td>
<td>iswctype(wc,wctype(&quot;alnum&quot;))</td>
</tr>
<tr>
<td>iswalpha(wc)</td>
<td>iswctype(wc,wctype(&quot;alpha&quot;))</td>
</tr>
<tr>
<td>iswcntrl(wc)</td>
<td>iswctype(wc,wctype(&quot;cntrl&quot;))</td>
</tr>
<tr>
<td>iswdigit(wc)</td>
<td>iswctype(wc,wctype(&quot;digit&quot;))</td>
</tr>
<tr>
<td>iswgraph(wc)</td>
<td>iswctype(wc,wctype(&quot;graph&quot;))</td>
</tr>
<tr>
<td>Function</td>
<td>Equivalent to locale specific test</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>iswlower(wc)</td>
<td>iswctype(wc,wctype(“lower”))</td>
</tr>
<tr>
<td>iswpunct(wc)</td>
<td>iswctype(wc,wctype(“punct”))</td>
</tr>
<tr>
<td>iswprint(wc)</td>
<td>iswctype(wc,wctype(“print”))</td>
</tr>
<tr>
<td>iswspace(wc)</td>
<td>iswctype(wc,wctype(“space”))</td>
</tr>
<tr>
<td>iswupper(wc)</td>
<td>iswctype(wc,wctype(“upper”))</td>
</tr>
<tr>
<td>iswxdigit(wc)</td>
<td>iswctype(wc,wctype(“xdigit”))</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Equivalent to locale specific transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>towlower(wc)</td>
<td>towctrans(wc,wctrans(“tolower”)</td>
</tr>
<tr>
<td>towupper(wc)</td>
<td>towctrans(wc,wctrans(“toupper”)</td>
</tr>
</tbody>
</table>
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

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

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

### Overview of string functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@CAT(str1,str2)</td>
<td>Concatenate strings</td>
</tr>
<tr>
<td>@LEN(string)</td>
<td>Length of string</td>
</tr>
<tr>
<td>@POS(str1,str2[,start])</td>
<td>Position of substring in string</td>
</tr>
<tr>
<td>@SCP(str1,str2)</td>
<td>Returns 1 if two strings are equal</td>
</tr>
<tr>
<td>@SUB(string,expr,expr)</td>
<td>Returns a substring</td>
</tr>
</tbody>
</table>

### Overview of macro functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@ARG('symbol'</td>
<td>expr)</td>
</tr>
<tr>
<td>@CNT()</td>
<td>Return number of macro arguments</td>
</tr>
<tr>
<td>@MAC(symbol)</td>
<td>Test if macro is defined</td>
</tr>
<tr>
<td>@MXP()</td>
<td>Test if macro expansion is active</td>
</tr>
</tbody>
</table>
Overview of address calculation functions

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

Overview of assembler mode functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@ASPCP()</td>
<td>Returns the name of the PCP assembler executable</td>
</tr>
<tr>
<td>@CPU(string)</td>
<td>Test if CPU type is selected</td>
</tr>
<tr>
<td>@DEF('symbol'</td>
<td>symbol)</td>
</tr>
<tr>
<td>@EXP(expr)</td>
<td>Expression check</td>
</tr>
<tr>
<td>@INT(expr)</td>
<td>Integer check</td>
</tr>
<tr>
<td>@LST()</td>
<td>LIST control flag value</td>
</tr>
</tbody>
</table>

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:

```assembly
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:

```assembly
.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:

```assembly
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:

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

@**COH**(*expression*)

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

Example:

```assembly
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
```
\textbf{@RVB(}\textit{expr1,expr2})

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

Example:
\begin{verbatim}
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
\end{verbatim}

\textbf{@SCP(}\textit{str1,str2})

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

Example:
\begin{verbatim}
.IF @SCP(STR,'MAIN'); does STR equal MAIN?
\end{verbatim}

\textbf{@SFRACT(}\textit{expression})

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

Example:
\begin{verbatim}
.WORD @SFRACT(0.1), @SFRACT(1.0)
\end{verbatim}

\textbf{@SGN(}\textit{expression})

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

Example:
\begin{verbatim}
VAR1 .SET @SGN(-1.2e-92); VAR1 = -1
VAR2 .SET @SGN(0); VAR2 = 0
VAR3 .SET @SGN(28.382); VAR3 = 1
\end{verbatim}

\textbf{@SIN(}\textit{expression})

Returns the sine of \textit{expression} as a floating-point value.

Example:
\begin{verbatim}
.WORD @SIN(@CVF(COUNT)*FREQ); compute sine value
\end{verbatim}
@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**

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.COMMENT</td>
<td>Start comment lines</td>
</tr>
<tr>
<td>.DEFINE</td>
<td>Define substitution string</td>
</tr>
<tr>
<td>.END</td>
<td>End of source program</td>
</tr>
<tr>
<td>.FAIL</td>
<td>Programmer generated error message</td>
</tr>
<tr>
<td>.INCLUDE</td>
<td>Include file</td>
</tr>
<tr>
<td>.MESSAGE</td>
<td>Programmer generated message</td>
</tr>
<tr>
<td>.ORG</td>
<td>Initialize memory space and location counters to</td>
</tr>
<tr>
<td></td>
<td>create a nameless section</td>
</tr>
<tr>
<td>.SDECL</td>
<td>Declare a section with name, type and attributes</td>
</tr>
<tr>
<td>.SECT</td>
<td>Activate a declared section</td>
</tr>
<tr>
<td>.UNDEF</td>
<td>Undefined DEFINE symbol</td>
</tr>
<tr>
<td>.WARNING</td>
<td>Programmer generated warning</td>
</tr>
</tbody>
</table>
### Overview of symbol definition directives

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

### Overview of data definition / storage allocation directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.ACCUM</td>
<td>Define 64-bit constant of 18 + 46 bits format</td>
</tr>
<tr>
<td>.ALIGN</td>
<td>Define alignment</td>
</tr>
<tr>
<td>.ASCII/.ASCIIZ</td>
<td>Define ASCII string without / with ending NULL byte</td>
</tr>
<tr>
<td>.BYTE</td>
<td>Define constant byte</td>
</tr>
<tr>
<td>.FLOAT/.DOUBLE</td>
<td>Define a 32-bit / 64-bit floating-point constant</td>
</tr>
<tr>
<td>.FRACT/.SFRACT</td>
<td>Define a 16-bit / 32-bit constant fraction</td>
</tr>
<tr>
<td>.SPACE</td>
<td>Define storage</td>
</tr>
<tr>
<td>.WORD/.HALF</td>
<td>Define a word / half-word constant</td>
</tr>
</tbody>
</table>
Overview of macro and conditional assembly directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.DUP / .ENDM</td>
<td>Duplicate sequence of source lines</td>
</tr>
<tr>
<td>.DUPA / .ENDM</td>
<td>Duplicate sequence with arguments</td>
</tr>
<tr>
<td>.DUPC / .ENDM</td>
<td>Duplicate sequence with characters</td>
</tr>
<tr>
<td>.DUPF / .ENDM</td>
<td>Duplicate sequence in loop</td>
</tr>
<tr>
<td>.EXITM</td>
<td>Exit macro</td>
</tr>
<tr>
<td>.IF / .ELIF / .ELSE / .ENDIF</td>
<td>Conditional assembly</td>
</tr>
<tr>
<td>.MACRO / .ENDM</td>
<td>Define macro</td>
</tr>
<tr>
<td>.PMACRO</td>
<td>Undefined (purge) macro definition</td>
</tr>
</tbody>
</table>

Overview of debug directives

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.CALLS</td>
<td>Passes call information to object file. Used by the linker to build a call graph and calculate stack size.</td>
</tr>
<tr>
<td>.MISRAC</td>
<td>Pass MISRA-C information</td>
</tr>
</tbody>
</table>

3.3.2 DETAILED DESCRIPTION OF ASSEMBLER DIRECTIVES

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

\texttt{\textit{alias-name} \texttt{.ALIAS} \textit{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 \texttt{.ALIAS} \texttt{\_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 section.

A label is not allowed before this directive.

Example

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

.ALIGN 12
add r5,r1,cc_uc
; WRONG: not a power of two, the
; 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 obtain 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:

\[
\text{.BYTE 'R'} ; = 0x52 \\
\text{.BYTE 'AB',,'D'} ; = 0x41420043
\]

Example

\[
\text{TABLE .BYTE 'two',0,'strings',0} \\
\text{CHARS .BYTE 'A','B','C','D'}
\]

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

\[ \text{[label]} \ .\text{DUP} \ expression \ .\text{ENDM} \]

Description

The sequence of source lines between the \texttt{.DUP} and \texttt{.ENDM} directives will be duplicated by the number specified by the integer \texttt{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 \texttt{.DUP} directive to any level.

If you specify \texttt{label}, it gets the value of the location counter at the start of the \texttt{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

- \texttt{.DUPA} (Duplicate Sequence with Arguments),
- \texttt{.DUPC} (Duplicate Sequence with Characters),
- \texttt{.DUPF} (Duplicate Sequence in Loop),
- \texttt{.MACRO} (Define Macro)
.DUAPA / .ENDM

Syntax

\[label\] .DUAPA formal_arg,argument[argument]...

.
.

.ENDM

Description

With the .DUAPA 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 .DUAPA directive processing.

Example

Consider the following source input statements,

```
.DUAPA 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

\[ [\text{label}] \ .\DUPC \ \text{formal_arg},\text{string} \ .\ .\ ENDM \]

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,

\[ .\DUPC \ \text{VALUE},'123' .\BYTE \ \text{VALUE} .\ENDM \]

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]

.DUPF

.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

\textit{symbol \ .EQU expression}

Description

With the \texttt{.EQU} directive you assign the value of \textit{expression} to \textit{symbol} permanently. Once defined, you cannot redefine the \textit{symbol}.

The \textit{expression} can be relocatable or absolute and forward references are allowed.

Example

To assign the value \texttt{0x4000} permanently to the symbol \texttt{A\_D\_PORT}:

\begin{verbatim}
A\_D\_PORT \ .EQU 0x4000
\end{verbatim}

You cannot redefine the symbol \texttt{A\_D\_PORT} after this.

Related information

\texttt{.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

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

```asm
.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

```asm
.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

[\textit{label}] \texttt{.FLOAT expression[,expression]...}

[\textit{label}] \texttt{.DOUBLE expression[,expression]...}

Description

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

An \textit{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 \textit{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

\texttt{FLT: .FLOAT 12.457,+0.27E-13}

\texttt{DBL: .DOUBLE 12.457,+0.27E-13}

Related information

\texttt{.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

```assembly
.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:

```assembly
.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
```
.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 alphanumerical 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:
```assembly
CONSTD .MACRO reg,value ;header
  ldl.iu reg,@hi(value) ;body
  ldl.il reg,@lo(value)
.ENDM ;terminator
```
The macro call:
```assembly
.SDECL '.pcptext', code
.SECT '.pcptext'
CONSTD r5,0x12345678
.END
```
The macro expands as follows:
```assembly
ldl.iu r5,@hi(0x12345678)
ldl.il r5,@lo(0x12345678)
```
### Related information
- .DUP (Duplicate Sequence of Source Lines),
- .DUAPA (Duplicate Sequence with Arguments),
- .DUPC (Duplicate Sequence with Characters),
- .DUPF (Duplicate Sequence in Loop)

.MESSAGE

Syntax

.MESSAGE [string | exp][string | exp]...

Description

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

.DEFINELONG "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 attributes:
  *init*  section is copied from ROM to RAM at startup
  *noread*  section can be executed from but not read

  Data attributes:
  *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 attributes:
- **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 attributes:

- **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

.WARNINDG  [(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

.WARNINDG  '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[,...]

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

\[ \text{label} \ .\text{WORD}\ \text{argument}[,\text{argument}]... \]
\[ \text{label} \ .\text{HALF}\ \text{argument}[,\text{argument}]... \]

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:

\[ .\text{WORD} \ 'R' \quad ; = 0x52 \]
\[ .\text{WORD} \ 'ABCD' \quad ; = 0x41424344 \]
\[ .\text{HALF} \ 'R' \quad ; = 0x52 \]
\[ .\text{HALF} \ 'AB' \quad ; = 0x4142 \]
\[ .\text{HALF} \ 'ABCD' \quad ; = 0x4142 \]
\[ \quad 0x4344 \]

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 obtain exactly the same effect:

```assembly
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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$LIST ON / OFF</td>
<td>Generation of assembly list file temporary ON/OFF</td>
</tr>
<tr>
<td>$LIST &quot;flags&quot;</td>
<td>Exclude / include lines in assembly list file</td>
</tr>
<tr>
<td>$PAGE</td>
<td>Generate formfeed in assembly list file</td>
</tr>
<tr>
<td>$PAGE settings</td>
<td>Define page layout for assembly list file</td>
</tr>
<tr>
<td>$PRCTL</td>
<td>Send control string to printer</td>
</tr>
<tr>
<td>$STITLE</td>
<td>Set program subtitle in header of assembly list file</td>
</tr>
<tr>
<td>$TITLE</td>
<td>Set program title in header of assembly list file</td>
</tr>
</tbody>
</table>

Overview of miscellaneous assembler controls

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$DEBUG ON / OFF</td>
<td>Generation of symbolic debug ON/OFF</td>
</tr>
<tr>
<td>$DEBUG &quot;flags&quot;</td>
<td>Select debug information</td>
</tr>
<tr>
<td>$HW_ONLY</td>
<td>Prevent substitution of assembly instructions by smaller or faster instructions</td>
</tr>
<tr>
<td>$IDENT LOCAL / GLOBAL</td>
<td>Assembler treats labels by default as local or global</td>
</tr>
<tr>
<td>$OBJECT</td>
<td>Alternative name for the generated object file</td>
</tr>
<tr>
<td>$WARNING OFF [num]</td>
<td>Suppress all or some warnings</td>
</tr>
</tbody>
</table>

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 $DEBUG OFF controls you turn the generation of debug information on or off. ($DEBUG ON is similar to the assembler option -g1).

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)
$\text{LIST ON } / \text{ OFF}$

Syntax

$\text{LIST ON}$

.. ; assembly source lines

$\text{LIST OFF}$

Description

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

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

; Counter value currently 1
$\text{LIST ON}$ ; Counter value = 2
$\text{LIST ON}$ ; Counter value = 3
$\text{LIST OFF}$ ; Counter value = 2
$\text{LIST OFF}$ ; Counter value = 1

The listing still would not be disabled until another $\text{LIST OFF}$ 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
$\text{LIST OFF}$
... ; source line not in list file
$\text{LIST ON}$
... ; source line also in list file
.END
```
The assembler generates a list file with the following lines:

```assembly
.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 Assembly 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 -Lflag, 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 Assembly 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 `-I` (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*.
$\text{PRCTL}$

Syntax

\[
\text{PRCTL} \ exp \mid \text{string}\{exp\mid \text{string}\}...
\]

Description

If you generate a list file with the assembler option -l, you can use the $\text{PRCTL}$ control to send control strings to the printer.

The $\text{PRCTL}$ 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:

\begin{itemize}
  \item \textit{exp} \quad \text{a byte expression which may be used to encode non-printing control characters, such as ESC.}
  \item \textit{string} \quad \text{an assembler string, which may be of arbitrary length, up to the maximum assembler-defined limits.}
\end{itemize}

The $\text{PRCTL}$ 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 $\text{PRCTL}$ 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 $\text{PRCTL}$ control to restore a printer to a previous mode after printing is done.

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

Example

\[
\text{PRCTL} \ \$1B,\'E' ; \text{Reset HP LaserJet printer}
\]

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 encounters a new STITLE control. By default, the subtitle is empty.

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

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

Example

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

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
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

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 suppress 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
4.1 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 chosen 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.

<table>
<thead>
<tr>
<th>Special value</th>
<th>Sign</th>
<th>Exponent</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0.0 (Positive Zero)</td>
<td>0</td>
<td>all zeros</td>
<td>all zeros</td>
</tr>
<tr>
<td>-0.0 (Negative Zero)</td>
<td>1</td>
<td>all zeros</td>
<td>all zeros</td>
</tr>
<tr>
<td>+INF (Positive Infinite)</td>
<td>0</td>
<td>all ones</td>
<td>all zeros</td>
</tr>
<tr>
<td>-INF (Negative Infinite)</td>
<td>1</td>
<td>all ones</td>
<td>all zeros</td>
</tr>
<tr>
<td>NaN (Not a number)</td>
<td>0</td>
<td>all ones</td>
<td>not all zeros</td>
</tr>
</tbody>
</table>

*Table 4-1: Special floating-point values*

### 4.5.3 TRAPPING FLOATING-POINT EXCEPTIONS

Four floating-point libraries are delivered:

<table>
<thead>
<tr>
<th>Library to link</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>libfp.a</td>
<td>Floating-point library (non-trapping). This is the default.</td>
</tr>
<tr>
<td>libfpt.a</td>
<td>Floating-point library (trapping)</td>
</tr>
<tr>
<td></td>
<td>(Control program option <code>--fp-trap</code>)</td>
</tr>
</tbody>
</table>

*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:

```c
#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:

```c
#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:

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

*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
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:

```c
  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
-\texttt{A[flags]}  
\texttt{--language=[flags]}

You can set the following flags:

\begin{itemize}
  \item \texttt{g/G} (+/-gcc) \quad \text{Enable a number of gcc extensions}
  \item \texttt{p/P} (+/-comments) \quad \text{Allow C++ style (/\*)comments in ISO C90}
  \item \texttt{x/X} (+/-strings) \quad \text{Relaxed const check for string literals}
\end{itemize}

Default
-\texttt{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 \texttt{-A (--language)} without flags is the equivalent of \texttt{-AGPX} and \textit{disables} all language extensions.

With \texttt{-Ag} you tell the compiler to enable the following gcc languages extensions:

\begin{itemize}
  \item The identifier \texttt{__FUNCTION__} expands to the current function name.
  \item Alternative syntax for variadic macros.
  \item Alternative syntax for designated initializers.
  \item Allow zero sized arrays.
  \item Allow empty struct/union.
  \item Allow empty initializer list.
  \item Allow initialization of static objects by compound literals.
  \item The middle operand of a \texttt{? :} operator may be omitted.
  \item Allow a compound statement inside braces as expression.
  \item Allow arithmetic on void pointers and function pointers.
  \item Allow a range of values after a single \texttt{case} label.
  \item Additional preprocessor directive \texttt{#warning}.
  \item Allow comma operator, conditional operator and cast as lvalue.
\end{itemize}
• An inline function without "\texttt{static}" or "\texttt{extern}" will be global.
• An "\texttt{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 (\texttt{info gcc}).

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

With \texttt{-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 produce a warning:

\begin{verbatim}
char *p;
void main( void ) { p = "hello"; }
\end{verbatim}

\textbf{Example}

\begin{verbatim}
cpcp -APx -c90 test.c
cpcp --language=-comments,+strings --iso=90 test.c
\end{verbatim}

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

\textbf{Related information}

\textbf{Compiler option \texttt{-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 because 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 `regcpu.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:

```
cp cp --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

Example

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:

```c
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:

```bash
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 (\texttt{--preprocess})

\textbf{Command line syntax}

\begin{verbatim}
-E[flags]

\texttt{--preprocess=[flags]}
\end{verbatim}

You can set the following flags (when you specify \texttt{-E} without flags, the default is \texttt{-ECMP}):

\begin{itemize}
  \item \texttt{c/C} (+/-comments) \hspace{2cm} Keep comments
  \item \texttt{m/M} (+/-make) \hspace{2cm} Generate dependencies for make
  \item \texttt{p/P} (+/-noline) \hspace{2cm} Strip \texttt{#}line source position info
\end{itemize}

\textbf{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 \texttt{-o}.

With \texttt{-Ec} you tell the preprocessor to keep the comments from the C source file in the preprocessed output.

With \texttt{-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 \texttt{-Ep} you tell the preprocessor to strip the \texttt{#}line source position information (lines starting with \texttt{#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.

\textbf{Example}

\begin{verbatim}
cpcp -EcMP test.c -o test.pre

cpcp --preprocess=+comments,-make,-noline test.c
   --output=test.pre
\end{verbatim}

The compiler preprocesses the file \texttt{test.c} and sends the output to the file \texttt{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 vice 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:

```bash
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
ncpcp --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

-\texttt{Ipath},...

--\texttt{include-directory=\texttt{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 \texttt{#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 \texttt{$\{PRODDIR\}/include} (unless you specified option --\texttt{no-stdinc}).

Example

Suppose that the C source file \texttt{test.c} contains the following lines:

\begin{verbatim}
#include <stdio.h>
#include "myinc.h"
\end{verbatim}

You can call the compiler as follows:

\begin{verbatim}
cpcp -Imyinclude test.c
cpcp --include-directory=myinclude test.c
\end{verbatim}

First the compiler looks for the file \texttt{stdio.h} in the directory \texttt{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 \texttt{myinc.h} in the directory where \texttt{test.c} is located. If the file is not there the compiler searches in the directory \texttt{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 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


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 -I myinclude --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 regcpu.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:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a/A</td>
<td>(+/-coalesce) Coalescer: remove unnecessary moves</td>
</tr>
<tr>
<td>b/B</td>
<td>(+/-ipro) Interprocedural register eliminations</td>
</tr>
<tr>
<td>c/C</td>
<td>(+/-cse) Common subexpression elimination</td>
</tr>
<tr>
<td>e/E</td>
<td>(+/-expression) Expressions simplification</td>
</tr>
<tr>
<td>f/F</td>
<td>(+/-flow) Control flow optimization and code reordering</td>
</tr>
<tr>
<td>g/G</td>
<td>(+/-glo) Generic assembly optimizations</td>
</tr>
<tr>
<td>h/H</td>
<td>(+/-partition) Automatic memory partition</td>
</tr>
<tr>
<td>i/I</td>
<td>(+/-inline) Function inlining</td>
</tr>
<tr>
<td>l/L</td>
<td>(+/-loop) Loop transformations</td>
</tr>
<tr>
<td>o/O</td>
<td>(+/-forward) Forward store</td>
</tr>
<tr>
<td>p/P</td>
<td>(+/-propagate) Constant propagation</td>
</tr>
<tr>
<td>r/R</td>
<td>(+/-compact) Code compaction (reverse inlining)</td>
</tr>
<tr>
<td>s/S</td>
<td>(+/-subscript) Subscript strength reduction</td>
</tr>
<tr>
<td>y/Y</td>
<td>(+/-peephole) Peephole optimizations</td>
</tr>
</tbody>
</table>

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: -OabcefgIIopRSy
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.

\[-O3 \quad (\text{--optimize=3})\]  
Aggressive optimization  
 Alias for: \([-OabcdefghiloprSy]\)

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 \[-OabcdefghiloprSy\]).

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
-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

-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

- \texttt{t2}

Description

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

Default: the compiler 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

-umacro_name
--undefinenamed=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

```bash
cpcp -v
```
```bash
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
```
-w (--no-warnings)

Command line syntax

-\texttt{w[\textit{nr}]}  
  \texttt{--no-warnings[=\textit{nr}]}

Description

With this option you can suppress 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 -\texttt{w} multiple times.

Example

To suppress all warnings:

\texttt{cpcp test.c -w}
\texttt{cpcp test.c --no-warnings}

To suppress warnings 135 and 136:

\texttt{cpcp test.c -w135 -w136}
\texttt{cpcp test.c --no-warnings=135 --no-warnings=136}

Related information

Compiler option \texttt{--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

- Ccpu

-- 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

-DMacro 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:

```
.IFDEF 'DEMO'

... ; 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

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

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

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>The default is plain text</td>
</tr>
<tr>
<td>html</td>
<td>Display explanation in HTML format</td>
</tr>
<tr>
<td>rtf</td>
<td>Display explanation in RTF format</td>
</tr>
</tbody>
</table>

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:

```bash
aspcp --diag=241
```

This results in the following message and explanation:

```markdown
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:

```bash
aspcp --diag=html:all > aserrors.html
```
Related information

-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 vice 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

-\( g [\text{flag}] \)

--debug-info[\text{flag}]

You can set the following flags:

- \( a/A \ (\pm-as\text{m}) \) Assembly source line information
- \( h/H \ (\pm-hl\text{l}) \) Pass HLL debug information
- \( l/L \ (\pm-local) \) Local symbols debug information
- \( s/S \ (\pm-smart) \) 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:

\[
\text{aspcp} \ -gAHLS \ test.src
\]

\[
\text{aspcp} \ --\text{debug-info}=-asm,\text{-hll,\text{-local,\text{-smart test.src}}}
\]

To enable smart debugging, enter:

\[
\text{aspcp} \ -gs \ test.src
\]

\[
\text{aspcp} \ --\text{debug-info}=+smart \ test.src
\]
Related information

-
-H (--include-file)

Command line syntax

- \texttt{H\{file,...\}}

\texttt{--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 \texttt{.INCLUDE 'file'} at the beginning of your assembly sources.

Example

\begin{verbatim}

aspcp -H myinc.inc test1.src
aspcp --include-file=myinc.inc test1.src
\end{verbatim}

The file \texttt{myinc.inc} is included at the beginning of \texttt{test1.src} before it is assembled.

Related information

\textbf{Assembler option -I (Add directory to include file search path)}

Section 5.6, \textit{How the Assembler Searches Include Files}, in Chapter \textit{Using the Assembler} of the User's Manual.
-l (--include-directory)

Command line syntax

-\texttt{l}\textit{path},...

\texttt{--include-directory}\textit{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 .\texttt{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 \texttt{test.src} contains the following line:

\texttt{.INCLUDE ‘myinc.inc’}

You can call the assembler as follows:

\texttt{aspcp -Ic:\proj\include test.src}
\texttt{aspcp --include-directory=c:\proj\include test.src}

First the assembler looks in the directory where \texttt{test.src} is located for the file \texttt{myinc.inc}. If it does not find the file, it looks in the directory \texttt{c:\proj\include} for the file \texttt{myinc.inc} (this option).

Related information

Section 5.6, \textit{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)
TOOL OPTIONS

-L (--list-format)

Command line syntax

-L flags

--list-format=flags

You can set the following flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
<th>Equivalent Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>same as -LCDEGILMNPQRVWXy</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>same as -Lcdegilmnpqrwxxy</td>
<td></td>
</tr>
<tr>
<td>d/D</td>
<td>(+/-section)</td>
<td>Section directives</td>
</tr>
<tr>
<td>e/E</td>
<td>(+/-symbol)</td>
<td>Symbol definition directives</td>
</tr>
<tr>
<td>g/G</td>
<td>(+/-generic-expansion)</td>
<td>Generic instruction expansion</td>
</tr>
<tr>
<td>i/I</td>
<td>(+/-generic)</td>
<td>Generic instructions</td>
</tr>
<tr>
<td>m/M</td>
<td>(+/-macro)</td>
<td>Macro definitions</td>
</tr>
<tr>
<td>n/N</td>
<td>(+/-empty-line)</td>
<td>Empty source lines</td>
</tr>
<tr>
<td>p/P</td>
<td>(+/-conditional)</td>
<td>Conditional assembly</td>
</tr>
<tr>
<td>q/Q</td>
<td>(+/-equate)</td>
<td>Assembler .EQU and .SET directives</td>
</tr>
<tr>
<td>r/R</td>
<td>(+/-relocations)</td>
<td>Relocation characters ('r')</td>
</tr>
<tr>
<td>v/V</td>
<td>(+/-equate-values)</td>
<td>Assembler .EQU and .SET values</td>
</tr>
<tr>
<td>w/W</td>
<td>(+/-wrap-lines)</td>
<td>Wrapped source lines</td>
</tr>
<tr>
<td>x/X</td>
<td>(+/-macro-expansion)</td>
<td>Macro expansions</td>
</tr>
<tr>
<td>y/Y</td>
<td>(+/-cycle-count)</td>
<td>Cycle counts</td>
</tr>
</tbody>
</table>

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)

-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

\[\text{aspcp test.src} \]
\[\text{aspcp -mt test.src} \]
\[\text{aspcp --preprocessor=tasking test.src} \]

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 \texttt{-C}).

With this option the assembler does not include the register file \texttt{regcpu.def} as based on the selected target processor.

Use this option if you want to use your own set of SFR '.def' files.

Example

\texttt{aspcp -Ctc1920b --no-tasking-sfr test.src}

The register file \texttt{regtc1920b.def} is not included, but the assembler allows the use of MMU instructions due to \texttt{-C}.

Related information

\textbullet\ Assembler option \texttt{-C} (Select CPU)
-O (--optimize)

Command line syntax

-flags

--optimize=flags

You can set the following flags:

\[ \begin{align*}
g/G & \ (\pm \text{generics}) & \text{Allow generic instructions} \\
s/S & \ (\pm \text{instr-size}) & \text{Optimize instruction size} \\
\end{align*} \]

Default

-0g

Description

With this option you can control the level of optimization. If you do not use this option, -Og is the default.

Example

The following invocations are equivalent and result all in the default optimizations:

\[
\begin{align*}
\text{aspcp} & \ \text{test.src} \\
\text{aspcp} & \ -0g \ \text{test.src} \\
\text{aspcp} & \ --\text{optimize}=+\text{generics},+\text{instr-size} \ \text{test.src}
\end{align*}
\]

Related information

Section 5.3, Assembler Optimizations, in Chapter Using the Assembler of the User’s Manual.
-o (--output)

Command line syntax

-ofile

--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

-Pprefix

--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 distinguish these symbols from symbols generated by the aspcp assembler.

Example

The add the prefix _PCP_ to global/external symbols, enter:

    aspcp -P_PCPC_test.pcp
    aspcp --prefix=_PCP_ test.pcp

Related information
-p (--pcptype)

Command line syntax

-ppcptype

--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

-flags

--section-info=flags

You can set the following flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c/C</td>
<td>(+/-console) Display section information on stdout.</td>
</tr>
<tr>
<td>l/L</td>
<td>(+/-list) Write section information to the list file.</td>
</tr>
</tbody>
</table>

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

- \texttt{-w[nr,...]}

\texttt{--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 \texttt{-w} multiple times.

Example

To suppress all warnings:

\begin{verbatim}
aspcp -w test.src
aspcp --no-warnings test.src
\end{verbatim}

To suppress warnings 135 and 136:

\begin{verbatim}
aspcp -w135,136 test.src
aspcp --no-warnings=135,136 test.src
\end{verbatim}

Related information

Assembler option \texttt{--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*,
**--case-insensitive**

**Command line syntax**

```bash
--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:

```bash
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:

```
lpcc 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 `tclv1_3.lsl`:

```
1pcp -dtclv1_3.lsl test.o
1pcp --lsl-file=tclv1_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):

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>The default is plain text</td>
</tr>
<tr>
<td>html</td>
<td>Display explanation in HTML format</td>
</tr>
<tr>
<td>rtf</td>
<td>Display explanation in RTF format</td>
</tr>
</tbody>
</table>

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

-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

-Im 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

(\texttt{--user-provided-initialization-code})

Command line syntax

- \texttt{-i}

- \texttt{--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 \textit{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 \texttt{--no-rom-copy} and \texttt{--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 (\texttt{--keep-output-files})

\textbf{Command line syntax}

\begin{verbatim}
-k
--keep-output-files
\end{verbatim}

\textbf{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 \texttt{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.

\textbf{Example}

\begin{verbatim}
lpcp -k test.o
lpcp --keep-output-files test.o
\end{verbatim}

When an error occurs during linking, the generated output file \texttt{test.elf} will \textit{not} be removed.

\textbf{Related information}
-L (--library-directory /
  --ignore-default-library-path)

Command line syntax

-\textit{L} \textit{path},...
  
  --library-directory=\textit{path},...

-\textit{L}
  
  --ignore-default-library-path

Description

With this option you can specify the path(s) where your system libraries, specified with the -\textit{L} option, are located. If you want to specify multiple paths, use the option -\textit{L} for each separate path.

The default path is $\$(PRODDIR)\lib$.

If you specify only -\textit{L} (without a pathname) or the long option
--ignore-default-library-path, the linker will \textit{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 -\textit{L} option is:

1. The path that is specified with the -\textit{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:

\texttt{lpcp test.o -Lc:\mylibs -lc}

First the linker looks in the directory \texttt{c:\mylibs} for library \texttt{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 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 `libname.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 `LIBTCL1V1_2`, `LIBTCL1V1_3` or `LIBTCL2`, 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)

---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:

<table>
<thead>
<tr>
<th>Number</th>
<th>Flags</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-mcfikLMNoQrSU</td>
<td>(link info)</td>
</tr>
<tr>
<td>1</td>
<td>-mCfiKlmNoQrSU</td>
<td>(locate info)</td>
</tr>
<tr>
<td>2</td>
<td>-mcfiklmNoQrSu</td>
<td>(most info)</td>
</tr>
<tr>
<td>c/C</td>
<td>(+/-callgraph)</td>
<td>Call graph info</td>
</tr>
<tr>
<td>f/F</td>
<td>(+/-files)</td>
<td>Processed files info</td>
</tr>
<tr>
<td>i/I</td>
<td>(+/-invocation)</td>
<td>Invocation and tool info</td>
</tr>
<tr>
<td>k/K</td>
<td>(+/-link)</td>
<td>Link result info</td>
</tr>
<tr>
<td>L/L</td>
<td>(+/-locate)</td>
<td>Locate result info</td>
</tr>
<tr>
<td>m/M</td>
<td>(+/-memory)</td>
<td>Memory usage info</td>
</tr>
<tr>
<td>n/N</td>
<td>(+/-nonalloc)</td>
<td>Non alloc info</td>
</tr>
<tr>
<td>o/O</td>
<td>(+/-overlay)</td>
<td>Overlay info</td>
</tr>
<tr>
<td>q/Q</td>
<td>(+/-statics)</td>
<td>Module local symbols</td>
</tr>
<tr>
<td>r/R</td>
<td>(+/-crossref)</td>
<td>Cross references info</td>
</tr>
<tr>
<td>s/S</td>
<td>(+/-lsl)</td>
<td>Processor and memory info</td>
</tr>
<tr>
<td>u/U</td>
<td>(+/-rules)</td>
<td>Locate rules</td>
</tr>
</tbody>
</table>

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:

```
1pcp --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

-Oflags

--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.

```
lpctest.o
lpctest
lpctest -O test.o
lpctest -O1 test.o
lpctest -OcLtxy test.o

lpctest --optimize test.o
lpctest --optimize=1 test.o
lpctest --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**

```
-ofile[[format][addr_size][,space_name]]...

--output=ofile[[format][addr_size][,space_name]]...
```

You can specify the following formats:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELF</td>
<td>ELF/DWARF</td>
</tr>
<tr>
<td>IEEE</td>
<td>IEEE-695</td>
</tr>
<tr>
<td>IHEX</td>
<td>Intel Hex</td>
</tr>
<tr>
<td>SREC</td>
<td>Motorola S-records</td>
</tr>
</tbody>
</table>

**Description**

By default, the linker generates an output file in ELF/DWARF format, with the name `task1.elf`.

With this option you can specify an alternative `filename`, and an alternative output `format`. The default output format is the format of the first input file.

You can use the `--output` option multiple times. This is useful to generate multiple output formats. With the first occurrence of the `--output` option you specify the basename (the filename without extension), which is used for subsequent `--output` options with no filename specified. If you do not specify a filename, or you do not specify the `--output` option at all, the linker uses the default basename `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 filled 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`:

```bash
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

- \texttt{r}
  \texttt{--incremental}

Description

Normally the linker links \textit{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 \texttt{.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 \texttt{.out}. The linker will now locate the file.

Example

In this example, the files \texttt{test1.o}, \texttt{test2.o} and \texttt{test3.o} are incrementally linked:

1. \texttt{lpcp -r test1.o test2.o -otest.out}
   \textit{test1.o and test2.o are linked}

2. \texttt{lpcp --incremental test3.o test.out}
   \textit{test3.o is linked together with test.out. File task1.out is created.}

3. \texttt{lpcp task1.out} \hspace{1cm} (\textit{task1.out is located})

Related information

-S (\texttt{--strip-debug})

\textbf{Command line syntax}

\begin{itemize}
\item \texttt{-S}
\item \texttt{--strip-debug}
\end{itemize}

\textbf{Description}

With this option you specify not to include symbolic debug information in the resulting output file.

\textbf{Example}

\begin{itemize}
\item lpcp \texttt{-S test.o -otest.elf}
\item lpcp \texttt{--strip-debug test.o --output=test.elf}
\end{itemize}

The linker generates the object file \texttt{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

-\(v[\text{\textbackslash v}]\)
  --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

\texttt{lpcp test.o -dextmem.lsl -deofault.lsl -lc -lfp -lrt -v}

The linker links the file test.o and displays the steps it performs.

\begin{verbatim}
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)
\end{verbatim}

Related information
### \(-w \text{ (--)no-warnings)}\)

**Command line syntax**

- \(-w[nr[,nr]...]\)
- \(--\text{no-warnings}[,nr]...\)

**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 warnings are suppressed. Separate multiple warnings by commas.

**Example**

To suppress all warnings:

```
lpccp -w test.o
lpccp --no-warnings test.o
```

To suppress warnings 113 and 114:

```
lpccp -w113,114 test.o
lpccp --no-warnings=113,114 test.o
```

**Related information**

- Linker option \(--\text{warnings-as-errors} \) (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

- Ccpu

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 program 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

-<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:

```c
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:

```bash
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)
\textbf{-d (\textendash \texttt{--lsl-file})}

\textbf{Command line syntax}

\begin{itemize}
  \item \texttt{-d\textasciitilde\textasciitilde lsl-file\textasciitilde\textasciitilde}
  \item \texttt{--lsl-file\textasciitilde\textasciitilde file}\textasciitilde\textasciitilde
\end{itemize}

\textbf{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:

\begin{itemize}
  \item the architecture and derivative definition describe the core’s hardware architecture and its internal memory.
  \item the board specification describes the physical memory available in the system.
  \item the section layout definition describes how to locate sections in memory.
\end{itemize}

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 $\texttt{tctarget.lsl}$ 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.

\textbf{Example}

To read linker script file information from file $\texttt{mylslfile.lsl}$:

\begin{verbatim}
ccpcp -dmylslfile.lsl test.obj
ccpcp --lsl-file=mylslfile.lsl test.obj
\end{verbatim}

\textbf{Related information}

**--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:

```
cpcp --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:

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

**Related information**

-
-E (--preprocess)

Command line syntax

-\[E[/flags]\]

\[--\text{preprocess}=[/flags]\]

You can set the following flags:

- \[c/C\] (+/-\text{comments})  Keep comments
- \[p/P\] (+/-\text{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

\[\text{ccpcp} -Ec \text{test.c} -o \text{test.pre}\]
\[\text{ccpcp} --\text{preprocess} +\text{comments},-\text{noline} \text{test.c}\]
\[--\text{output=}\text{test.pre}\]

The compiler preprocesses the file \text{test.c} and sends the output to the file \text{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**

```plaintext
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
```
**--format**

**Command line syntax**

```
--format=format
```

You can specify the following formats:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELF</td>
<td>ELF/DWARF</td>
</tr>
<tr>
<td>IHEX</td>
<td>Intel Hex</td>
</tr>
<tr>
<td>SREC</td>
<td>Motorola S-records</td>
</tr>
</tbody>
</table>

**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

-IPath

--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:

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

You can call the control program as follows:

```c
ccpcp -Imyinclude test.c
ccpcp --include-directory= myinclude
```

First the compiler looks for the file stdout.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

-ldpath
  --library-directory=dpath

-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 libname)
-l (--library)

Command line syntax

-lname

--library=name

Description

With this option you tell the linker via the control program to search also in system library libname.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 LIBTCLV1_2, LIBTCLV1_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 -lf
    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)

--list-files

Command line syntax

-`lname`

--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 `-llibrary_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 unresolved 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 -Ctcllib --no-tasking-sfr test.c

The register file regtcllib.sfr is not included.

Related information

Compiler option -C (Use SFR definitions for CPU)
-o (--output)

Command line syntax

-ofile

--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 (\texttt{--keep-temporary-files})

Command line syntax

- \texttt{t}

\texttt{--keep-temporary-files}

Description

By default, the control program removes intermediate files like the \texttt{.src} file (result of the compiler phase) and the \texttt{.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:

\begin{verbatim}
ccpcp -t test.c
ccpcp --keep-temporary-files test.c
\end{verbatim}

The control program keeps all intermediate files it generates while creating the absolute object file \texttt{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 Program 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
\textbf{\texttt{--v (\texttt{--verbose})}}

\textbf{Command line syntax}

\begin{verbatim}
   -v
   --verbose
\end{verbatim}

\textbf{Description}

With this option you put the control program in \textit{verbose} mode. With the option \texttt{--v} the control program performs its tasks while it prints the steps it performs to \texttt{stdout}.

\textbf{Example}

\begin{verbatim}
   ccpcp -v test.c
   ccpcp --verbose test.c
\end{verbatim}

The control program processes the file \texttt{test.c} and displays the invocations of the tools it uses to create the final object file.

\textbf{Related information}

Control program option \texttt{-n} (Verbose output and suppress execution)
**-Wtool (--pass)**

**Command line syntax**

- `Wcoption --pass-c=option`  
  Pass option directly to the C compiler
- `Waoption --pass-assembler=option`  
  Pass option directly to the assembler
- `Wloption --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

-\texttt{w}[nr]
-\texttt{--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:

\begin{verbatim}
ccpcp -w test.c
ccpcp --no-warnings test.c
\end{verbatim}

To suppress warnings 100:

\begin{verbatim}
ccpcp -w100 test.c
ccpcp --no-warnings=100 test.c
\end{verbatim}

Related information

Control program option \texttt{--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

\textit{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 \texttt{-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 \textit{option file} which you then must specify to the compiler with the option \texttt{-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:

\begin{verbatim}
ifdef DEMO    # the value of DEMO is of no importance
   real.out : demo.o
              lpcp demo.o main.o -lc -lfp -lrt
endif

real.elf : real.out
           lpcp -FELF -oreal.elf real.out
\end{verbatim}

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

\begin{verbatim}
mkpcp real.elf DEMO=1
\end{verbatim}

In both cases the absolute object file \texttt{real.elf} is created but depending on the DEMO flag it is linked with \texttt{demo.o} or with \texttt{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:

make –?

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**

`mkcp -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
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)
Command line syntax

- \texttt{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 \texttt{-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 vice 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 \texttt{\} to continue the line. Whitespace between quotes is preserved.

  "This is a continuation \ line"

  \rightarrow "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)
\textbf{\texttt{-p}}

\textbf{Command line syntax}

\texttt{-p}

\textbf{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.

\textbf{Example}

\begin{verbatim}
  mkpcep -p
\end{verbatim}

The make utility never removes target dependency files.

\textbf{Related information}

Special target .PRECIOUS in section 7.3.2, \textit{Writing a Makefile} in Chapter \textit{Using the Utilities} of the \textit{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
\textbf{-r}

\textbf{Command line syntax}

\texttt{-r}

\textbf{Description}

When you call the make utility, it first reads the implicit rules from the file \texttt{mkpcp.mk}, then it reads the makefile with the rules to build your files. (The file \texttt{mkpcp.mk} is located in the \texttt{etc} directory of the PCP toolchain.)

With this option you tell the make utility \textit{not} to read \texttt{mkpcp.mk} and to rely fully on the make rules in the makefile.

\textbf{Example}

\texttt{mkpcp -r}

The make utility does not read the implicit make rules in \texttt{mkpcp.mk}.

\textbf{Related information}

\begin{itemize}
  \item 
\end{itemize}
-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)
\textbf{-S}

\textbf{Command line syntax}

\texttt{-s}

\textbf{Description}

With this option you tell the make utility to perform its tasks without printing the commands it executes. Error messages are normally printed.

\textbf{Example}

\texttt{mkpcp -s}

The make utility rebuilds your files but does not print the commands it executes during the make process.

\textbf{Related information}

Make utility option \texttt{-n} (Perform a dry run)
\texttt{-t}

\textbf{Command line syntax}

\texttt{-t}

\textbf{Description}

With this option you tell the make utility to \textit{touch} the target files, bringing them up to date, rather than performing the rules to rebuild them.

\textbf{Example}

\texttt{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.

\textbf{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

-\texttt{w}

Description

With this option the make utility sends error messages and verbose messages to standard output. Without this option, the make utility sends these messages to standard error.

This option is only useful on UNIX systems.

Example

\texttt{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.

<table>
<thead>
<tr>
<th>Description</th>
<th>Option</th>
<th>Suboption</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display an overview of all options</td>
<td>-?</td>
<td></td>
</tr>
<tr>
<td>Display version information</td>
<td>-V</td>
<td></td>
</tr>
<tr>
<td>Read options from file</td>
<td>-f file</td>
<td></td>
</tr>
<tr>
<td>Print object module to standard output</td>
<td>-p</td>
<td></td>
</tr>
<tr>
<td>Suppress warnings above level n</td>
<td>-wn</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Main functions</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Delete object module from library</td>
<td>-d</td>
<td>-v</td>
</tr>
<tr>
<td>Move object module to another position</td>
<td>-m</td>
<td>-a -b -v</td>
</tr>
<tr>
<td>Replace or add an object module</td>
<td>-r</td>
<td>-a -b -c -u -v</td>
</tr>
<tr>
<td>Print a table of contents of the library</td>
<td>-t</td>
<td>-s0 -s1</td>
</tr>
<tr>
<td>Extract an object module from the library</td>
<td>-x</td>
<td>-v</td>
</tr>
</tbody>
</table>

*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:

```
arpccp -?
arpccp
```

**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

arcp -d lib.a obj1.o obj2.o

The archiver deletes obj1.o and obj2.o from the library lib.a.

arcp -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

-ffile

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"

- 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
```
\textbf{\texttt{\textasciitilde m}}

**Command line syntax**

\texttt{\textasciitilde 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 \texttt{-a} or \texttt{-b} to move them to a specified place instead.

- \texttt{-a posname} Move the specified object module(s) after the existing module \textit{posname}.
- \texttt{-b posname} Move the specified object module(s) before the existing module \textit{posname}.

**Example**

Suppose the library \texttt{lib.a} contains the following objects (see option \texttt{-t}):

\begin{verbatim}
obj1.o
obj2.o
obj3.o
\end{verbatim}

To move \texttt{obj1.o} to the end of \texttt{lib.a}:

\begin{verbatim}
arpcp -m lib.a obj1.o
\end{verbatim}

To move \texttt{obj3.o} just before \texttt{obj2.o}:

\begin{verbatim}
arpcp -m -b obj3.o lib.a obj2.o
\end{verbatim}

The library \texttt{lib.a} after these two invocations now looks like:

\begin{verbatim}
obj3.o
obj2.o
obj1.o
\end{verbatim}

**Related information**

- Archiver option \texttt{-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

arcp -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 \(-t\)):

\[
\text{obj1.o}
\]

To add \text{obj2.o} to the end of \text{lib.a}:

\text{arpcp -r lib.a obj2.o}

To insert \text{obj3.o} just before \text{obj2.o}:

\text{arpcp -r -b obj2.o lib.a obj3.o}

The library \text{lib.a} after these two invocations now looks like:

\[
\text{obj1.o} \\
\text{obj3.o} \\
\text{obj2.o}
\]

Creating a new library

To \textit{create a new library file}, add an object file and specify a library that does not yet exist:

\text{arpcp -r obj1.o newlib.a}

The archiver creates the library \texttt{newlib.a} and adds the object \texttt{obj1.o} to it.

To \textit{create a new library file and overwrite an existing library}, add an object file and specify an existing library with the supoption \texttt{-c}:

\text{arpcp -r -c obj1.o lib.a}

The archiver overwrites the library \texttt{lib.a} and adds the object \texttt{obj1.o} to it. The new library \texttt{lib.a} only contains \texttt{obj1.o}.

Related information

\text{Archiver option \texttt{-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**
\textbf{-V}

\textbf{Command line syntax}

\texttt{-V}

\textbf{Description}

Display version information. The archiver ignores all other options or input files.

\textbf{Example}

\texttt{arpcp -V}

The archiver does not perform any tasks but displays the following version information:

\begin{verbatim}
TASKING PCP VX-toolset ELF archiver vxx.yrz Build nnn
Copyright year Altium BV Serial# 00000000
\end{verbatim}
-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

–wlevel

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

-
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
```

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      ld1.i1    r7,@DPTR(_PCP__99900...
0002    54rr     27      st.pi     r2,[_PCP__999001__1]   
0003    9340rrrr  28      ld1.i1    r5,@LO(_PCP__lc_ub__...
0005    93C0rrrr  29      ld1.i1    r7,@DPTR(_PCP__data__...
0007    55rr     30      st.pi     r5,[_PCP__data__printf]
0008    9240rrrr  31      ld1.i1    r1,@LO(_PCP__1_str)   
000A    93C0rrrr  32      ld1.i1    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**

```
+---------------------------------------------------------+
<table>
<thead>
<tr>
<th>File</th>
<th>From archive</th>
<th>Symbol causing the extraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>cstart.o</td>
<td>libc.a</td>
<td>_START</td>
</tr>
<tr>
<td>hello.o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>printf.o</td>
<td>libc.a</td>
<td>printf</td>
</tr>
<tr>
<td>----------</td>
<td>--------------</td>
<td>------------------------------</td>
</tr>
</tbody>
</table>
```

```
+-----------------------------------------------------------------------------+
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>hello.o</td>
<td>.text.hello.main</td>
<td>0x00000014</td>
<td>0x00000000</td>
<td>.text.hello.main</td>
</tr>
<tr>
<td>cstart.o</td>
<td>.text.libc</td>
<td>0x00000202</td>
<td>0x00000000</td>
<td>.text.libc</td>
</tr>
<tr>
<td>strcpy.o</td>
<td>.text.libc</td>
<td>0x00000024</td>
<td>0x00000204</td>
<td></td>
</tr>
<tr>
<td>printf.o</td>
<td>.text.libc</td>
<td>0x0000002c</td>
<td>0x00000800</td>
<td></td>
</tr>
<tr>
<td>cstart.o</td>
<td>.text.libc.reset</td>
<td>0x00000008</td>
<td>0x00000000</td>
<td>.text.libc.reset</td>
</tr>
</tbody>
</table>
|----------------------------------------------------------------------+
```

```
* Symbol translation (sorted on name)

+ Scope "/hello.o"

```
+-----------------------------------------------+
<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>hello.src</td>
<td>0x00000000</td>
<td></td>
</tr>
<tr>
<td>.rodata_hello</td>
<td>0xa000000e</td>
<td>spe:tc:linear</td>
</tr>
<tr>
<td>.text_hello.main</td>
<td>0xa00000f8</td>
<td>spe:tc:abs18</td>
</tr>
<tr>
<td>.zdata_hello</td>
<td>0xd0000000</td>
<td>spe:tc:abs18</td>
</tr>
<tr>
<td>.zrodata_hello</td>
<td>0xa0000008</td>
<td>spe:tc:abs18</td>
</tr>
</tbody>
</table>
+-----------------------------------------------+
```
Cross Reference Part

<table>
<thead>
<tr>
<th>Definition file</th>
<th>Definition section</th>
<th>Symbol</th>
<th>Referenced in</th>
</tr>
</thead>
<tbody>
<tr>
<td>_doprint_int.o</td>
<td>.text.libc</td>
<td>__printf_int2</td>
<td>printf_int.o</td>
</tr>
<tr>
<td>_doprint_int.o</td>
<td>.text.libc</td>
<td>_doprint</td>
<td>printf.o</td>
</tr>
<tr>
<td>cstart.o</td>
<td>.text.libc</td>
<td>_start</td>
<td>hello.o</td>
</tr>
<tr>
<td>cstart.o</td>
<td>.text.libc.reset</td>
<td>_START</td>
<td></td>
</tr>
<tr>
<td>hello.o</td>
<td>.text.hello.main</td>
<td>main</td>
<td>cstart.o</td>
</tr>
<tr>
<td>printf.o</td>
<td>.text.libc</td>
<td>printf</td>
<td>hello.o</td>
</tr>
</tbody>
</table>

Call Graph Part

Overlay Part

Locate Part

Task entry address

<table>
<thead>
<tr>
<th>symbol</th>
<th>_START</th>
</tr>
</thead>
<tbody>
<tr>
<td>absolute addr</td>
<td>0xa0000000</td>
</tr>
</tbody>
</table>

Section translation

Space spe:tc:linear

<table>
<thead>
<tr>
<th>Chip</th>
<th>Group</th>
<th>Section</th>
<th>Size (MAU)</th>
<th>Space addr</th>
<th>Chip addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>ext_c</td>
<td></td>
<td>.text.libc.reset</td>
<td>0x00000008</td>
<td>0xa0000000</td>
<td>0x00000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[.data.libc]</td>
<td>0x000000cc</td>
<td>0xa0000010</td>
<td>0x00000010</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[.zdata.hello]</td>
<td>0x00000004</td>
<td>0xa00000dc</td>
<td>0x000000dc</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.rodata.hello</td>
<td>0x0000000c</td>
<td>0xa00000ec</td>
<td>0x000000ec</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.rodata.libc</td>
<td>0x0000000c</td>
<td>0xa00000ec</td>
<td>0x000000ec</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.text.hello.main</td>
<td>0x00000014</td>
<td>0xa00000f8</td>
<td>0x000000f8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.text.libc.csa_areas</td>
<td>0x00000008</td>
<td>0xa000010c</td>
<td>0x0000010c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>table</td>
<td>0x00000034</td>
<td>0xa0000114</td>
<td>0x00000114</td>
</tr>
<tr>
<td></td>
<td>libraries</td>
<td>.text.libc</td>
<td>0x0000eae</td>
<td>0xa0004000</td>
<td>0x00004000</td>
</tr>
</tbody>
</table>

Space spe:tc:abs18

<table>
<thead>
<tr>
<th>Chip</th>
<th>Group</th>
<th>Section</th>
<th>Size (MAU)</th>
<th>Space addr</th>
<th>Chip addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>ext_c</td>
<td></td>
<td>.zrodata.hello</td>
<td>0x00000006</td>
<td>0xa0000000</td>
<td>0x00000000</td>
</tr>
<tr>
<td>spe:dsram</td>
<td></td>
<td>.zdata.hello</td>
<td>0x00000004</td>
<td>0xd0000000</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>

Symbol translation (sorted on name)

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>A8_DATA</em></td>
<td>0x00000000</td>
<td>spe:tc:linear</td>
</tr>
<tr>
<td>_Exit</td>
<td>0xa0004170</td>
<td></td>
</tr>
<tr>
<td>_START</td>
<td>0xa0000000</td>
<td></td>
</tr>
<tr>
<td>_lc_gb_int_tab</td>
<td>0x00000000</td>
<td></td>
</tr>
<tr>
<td>_lc_ge_int_tab</td>
<td>0x00000000</td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>0xa000000f8</td>
<td></td>
</tr>
<tr>
<td>world</td>
<td>0xd0000000</td>
<td>spe:tc:abs18</td>
</tr>
</tbody>
</table>
* Symbol translation (sorted on address)

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>_lc_ge_int_tab</td>
<td>spe:tc:linear</td>
</tr>
<tr>
<td>0x00000000</td>
<td>_lc_gb_int_tab</td>
<td></td>
</tr>
<tr>
<td>0x00000000</td>
<td><em>A8_DATA</em></td>
<td></td>
</tr>
<tr>
<td>0xa0000000</td>
<td>_START</td>
<td></td>
</tr>
<tr>
<td>0xa00000f8</td>
<td>main</td>
<td></td>
</tr>
<tr>
<td>0xa0004170</td>
<td>_Exit</td>
<td></td>
</tr>
<tr>
<td>0xd0000000</td>
<td>world</td>
<td>spe:tc:abs18</td>
</tr>
</tbody>
</table>

* Address range usage at space level

<table>
<thead>
<tr>
<th>Name</th>
<th>Total</th>
<th>Used        %</th>
<th>Free        %</th>
<th>&gt; free gap  %</th>
</tr>
</thead>
<tbody>
<tr>
<td>spe:tc:abs18</td>
<td>0x00008000</td>
<td>0x00000146   1</td>
<td>0x00007eba  99</td>
<td>0x00004000  50</td>
</tr>
<tr>
<td>spe:tc:abs24</td>
<td>0x50024000</td>
<td>0x00022820   1</td>
<td>0x500017e0  99</td>
<td>0x2ff0df0a  59</td>
</tr>
<tr>
<td>spe:ccsa</td>
<td>0x00006000</td>
<td>0x00001004  17</td>
<td>0x00004ffc  83</td>
<td>0x00004fc0  83</td>
</tr>
<tr>
<td>spe:clinear</td>
<td>0x00008500</td>
<td>0x00021788  26</td>
<td>0x00001788  99</td>
<td>0x00004000  50</td>
</tr>
<tr>
<td>spe:pcp_code</td>
<td>0x00002000</td>
<td>0x00000000   0</td>
<td>0x00002000 100</td>
<td>0x00002000 100</td>
</tr>
<tr>
<td>spe:pcp_data</td>
<td>0x00000400</td>
<td>0x00000000   0</td>
<td>0x00000400 100</td>
<td>0x00000400 100</td>
</tr>
</tbody>
</table>

* Address range usage at memory level

<table>
<thead>
<tr>
<th>Name</th>
<th>Total</th>
<th>Used        %</th>
<th>Free        %</th>
<th>&gt; free gap  %</th>
</tr>
</thead>
<tbody>
<tr>
<td>ext_c</td>
<td>0x00080000</td>
<td>0x00000ff4   1</td>
<td>0x0007f00c  99</td>
<td>0x0007b152  96</td>
</tr>
<tr>
<td>ext_c2</td>
<td>0x00080000</td>
<td>0x00000000   0</td>
<td>0x00080000 100</td>
<td>0x00080000 100</td>
</tr>
<tr>
<td>ext_d</td>
<td>0x00070000</td>
<td>0x00020784  29</td>
<td>0x0004f87c  71</td>
<td>0x0004f878  71</td>
</tr>
<tr>
<td>spec:brom</td>
<td>0x00008000</td>
<td>0x00000000   0</td>
<td>0x00008000 100</td>
<td>0x00008000 100</td>
</tr>
<tr>
<td>spec:csram</td>
<td>0x00006000</td>
<td>0x00000000   0</td>
<td>0x00006000 100</td>
<td>0x00006000 100</td>
</tr>
<tr>
<td>spec:dsram</td>
<td>0x00006000</td>
<td>0x00001004  17</td>
<td>0x00004ffc  83</td>
<td>0x00004fc0  83</td>
</tr>
<tr>
<td>spec:fpidram</td>
<td>0x00004000</td>
<td>0x00000000   0</td>
<td>0x00004000 100</td>
<td>0x00004000 100</td>
</tr>
<tr>
<td>spec:code</td>
<td>0x00004000</td>
<td>0x00000000   0</td>
<td>0x00004000 100</td>
<td>0x00004000 100</td>
</tr>
<tr>
<td>spec:pram</td>
<td>0x00001000</td>
<td>0x00000000   0</td>
<td>0x00001000 100</td>
<td>0x00001000 100</td>
</tr>
<tr>
<td>vecttable</td>
<td>0x00002400</td>
<td>0x000000a4   2</td>
<td>0x0000235c  98</td>
<td>0x00002000  88</td>
</tr>
</tbody>
</table>

* Linker Script File Part

* Locate Rule Part

<table>
<thead>
<tr>
<th>Address space</th>
<th>Type</th>
<th>Properties</th>
<th>Sections</th>
</tr>
</thead>
<tbody>
<tr>
<td>spe:tc:linear</td>
<td>absolute</td>
<td>0xa0004000</td>
<td>.text.libc</td>
</tr>
<tr>
<td>spe:tc:abs18</td>
<td>clustered</td>
<td></td>
<td>.zdata.libc</td>
</tr>
<tr>
<td>spe:tc:linear</td>
<td>clustered</td>
<td></td>
<td>.data.libc</td>
</tr>
<tr>
<td>spe:tc:linear</td>
<td>ordered</td>
<td></td>
<td>.text.trapvec.000 &lt; .text.trapvec.001 ...</td>
</tr>
<tr>
<td>spe:tc:abs18</td>
<td>unrestricted</td>
<td></td>
<td>.zrodata.lib</td>
</tr>
<tr>
<td>spe:tc:linear</td>
<td>unrestricted</td>
<td></td>
<td>usock</td>
</tr>
</tbody>
</table>
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 addresses are not included in the range.

address range size

The sections must be located in some address range with size not larger than shown in the Properties column; the second number in that field is the alignment requirement for the address range.

ballooned

After locating all sections, the largest remaining gap in the space is used completely for the stack and/or heap.

Properties

The contents depend 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
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 `-ofilename:SREC` option the linker produces output in Motorola S-record format with three types of S-records: S0, S2 and S8. With the options `-ofilename:SREC:2` or `-ofilename: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

```
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 `-ofilename: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 `-ofilename: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:

```
S213FF0020023222754E00754F04AF4FAE4E22BF
      |   |  |     |_ 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 `-ofilename: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 `-ofilename: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 `-ofilename: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 `-ofilename: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 `-ofilename: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:
<table>
<thead>
<tr>
<th>Byte Type</th>
<th>Record type</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Data</td>
</tr>
<tr>
<td>01</td>
<td>End of File</td>
</tr>
<tr>
<td>02</td>
<td>Extended segment address (not used)</td>
</tr>
<tr>
<td>03</td>
<td>Start segment address (not used)</td>
</tr>
<tr>
<td>04</td>
<td>Extended linear address (32-bit)</td>
</tr>
<tr>
<td>05</td>
<td>Start linear address (32-bit)</td>
</tr>
</tbody>
</table>

\textit{content} is the information contained in the record. This depends on the record type.

\textit{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:

\begin{center}
\begin{tabular}{|c|c|}
\hline
: & 02 \hspace{0.5cm} 0000 \hspace{0.5cm} 04 \hspace{0.5cm} upper\_address \hspace{0.5cm} checksum \\
\hline
\end{tabular}
\end{center}

The 32-bit absolute address of a byte in a Data Record is calculated as:

\[( \text{address} + \text{offset} + \text{index} ) \text{ modulo } 4G\]

where:

\textit{address} is the base address, where the two most significant bytes are the \textit{upper\_address} and the two least significant bytes are zero.

\textit{offset} is the 16-bit offset from the Data Record.

\textit{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.

<table>
<thead>
<tr>
<th>:</th>
<th>length</th>
<th>offset</th>
<th>00</th>
<th>data</th>
<th>checksum</th>
</tr>
</thead>
</table>

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:

:040000500FF0003F5

| | | | | _ checksum
| | | | _ address
| | | _ type
| | _ offset
| _ length

**End of File Record**

The hexadecimal file always ends with the following end-of-file record:

:0000001FF

| | | _ checksum
| | _ type
| _ offset
| _ length
CHAPTER 8

LINKER SCRIPT
LANGUAGE
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.

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:

```plaintext
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 preprocessor. 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:

```plaintext
#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 $B$
$$<B>^0\{0$$ = zero of more occurrences of $B$
$$<B>^0\{1$$ = 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.

\[
\text{number} ::= \text{OCT\_NUM} \\
\quad \text{DEC\_NUM} \\
\quad \text{HEX\_NUM}
\]

\[
\text{expr} ::= \text{number} \\
\quad \text{symbol\_name} \\
\quad \text{unary\_op} \text{expr} \\
\quad \text{expr} \text{binary\_op} \text{expr} \\
\quad \text{expr} \text{?} \text{expr} : \text{expr} \\
\quad (\text{expr}) \\
\quad \text{function\_call}
\]

\[
\text{unary\_op} ::= ! \quad \text{// logical NOT} \\
\quad \text{-} \quad \text{// bitwise complement} \\
\quad \text{-} \quad \text{// negative value}
\]

\[
\text{binary\_op} ::= ^ \quad \text{// exclusive OR} \\
\quad \ast \quad \text{// multiplication} \\
\quad / \quad \text{// division} \\
\quad \% \quad \text{// modulus} \\
\quad + \quad \text{// addition} \\
\quad - \quad \text{// subtraction} \\
\quad >> \quad \text{// right shift} \\
\quad << \quad \text{// left shift} \\
\quad == \quad \text{// equal to} \\
\quad != \quad \text{// not equal to} \\
\quad > \quad \text{// greater than} \\
\quad < \quad \text{// less than} \\
\quad \geq \quad \text{// greater than or equal to} \\
\quad \leq \quad \text{// less than or equal to} \\
\quad \& \quad \text{// bitwise AND} \\
\quad | \quad \text{// bitwise OR} \\
\quad && \quad \text{// logical AND} \\
\quad || \quad \text{// logical OR}
\]
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()  

```c
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:

```c
exists( "mysection" )
```

max()  

```c
int max( expr, expr )
```

Returns the value of the expression that has the largest value. To get the highest value of two symbols:

```c
max( "sym1", "sym2")
```

min()  

```c
int min( expr, expr )
```

Returns the value of the expression hat has the smallest value. To get the lowest value of two symbols:

```c
min( "sym1", "sym2")
```

sizeof()  

```c
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":

```c
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 ::= \langle definition\rangle >= 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
             | 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:**).

\[
\text{mapping} \quad ::= \text{map} \left( \text{map_descr} <, \text{map_descr} >>=0 \right)
\]

\[
\text{map_descr} \quad ::= \quad \text{dest} = \text{destination} \\
\quad | \quad \text{dest_dbits} = \text{range} \\
\quad | \quad \text{dest_offset} = \text{expr} \\
\quad | \quad \text{size} = \text{expr} \\
\quad | \quad \text{src_dbits} = \text{range} \\
\quad | \quad \text{src_offset} = \text{expr}
\]

- 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.

\[
\text{destination} \quad ::= \quad \text{space} : \text{space\_name} \\
\quad | \quad \text{bus} : <\text{proc\_name} | \text{core\_name} >|_0|_1 \text{bus\_name}
\]

- 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

\[
\text{range} \quad ::= \quad \text{number} .. \text{number}
\]
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 ]
| page =< | [ range ]>=0 >0|1
| 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.

\[
\text{copy_table descr} \ := \ \text{align} = \text{expr} \\
\quad | \ \text{copy_unit} = \text{expr} \\
\quad | \ \text{dest} <\text{space_name}>0|1 = \text{space_name} \\
\quad | \ \text{page}
\]

• 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.

\[
\text{start_addr} \ := \ \text{start_address} ( \ \text{start_addr descr} \\
\quad <<, \ \text{start_addr descr}>>=0 )
\]

\[
\text{start_addr descr} \ := \ \text{run_addr} = \text{expr} \\
\quad | \ \text{symbol} = \text{symbol_name}
\]

• A symbol_name refers to the section that contains the startup code.

\[
\text{endianness_def} \ := \ \text{endianness} \{ \ <\text{endianness_type};>>=1 \}
\]

\[
\text{endianness_type} \ := \ \text{big} \\
\quad | \ \text{little}
\]

### 8.3.9 DERIVATIVE DEFINITION

\[
\text{derivative_definition} \ := \ \text{derivative} \ \text{derivative_name} \\
\quad \{ \ <\text{parameter_list }>>=0|1 \} \\
\quad \text{extends} \ \text{derivative_name} \\
\quad \{ \ <\text{argument_list }>>=0|1 >0|1 \} \\
\quad \{ \ <\text{derivative_spec }>>=0 \}
\]

• A derivative_definition defines a derivative with the given derivative_name as a unique name.

\[
\text{derivative_spec} \ := \ \text{core_def} \\
\quad | \ \text{bus_def} \\
\quad | \ \text{mem_def} \\
\quad | \ \text{section_definition} // no processor \\
\quad \text{// name}
\]

\[
\text{core_def} \ := \ \text{core} \ \text{core_name} \{ \ <\text{descr }>>=0 \}
\]
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`.

- 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

- 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

- 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


```plaintext

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>[^0,1]

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 ] |[^0,1]

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[^0,1]
```

**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 or little)
- `bus`
  - `mau`
  - `width`
  - `map`
- `space`
- `id`
- `mau`
- `align`
- `page_size`
- `page`
- `direction` (low_to_high or high_to_low)
- `stack`
  - `min_size`
  - `grows` (low_to_high or high_to_low)
  - `align`
  - `fixed`
- `heap`
  - `min_size`
  - `grows` (low_to_high or high_to_low)
  - `align`
  - `fixed`
- `copytable`
  - `align`
  - `copy_unit`
- `dest`
- `page`
- `start_address`
- `run_addr`
- `symbol`
- `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*.

```plaintext
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.

```plaintext
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.

```plaintext
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:

```plaintext
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.

```plaintext
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:

```markdown
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:

```plaintext
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.

```plaintext
derivative name_child_deriv (parm1, parm2=1)
    extends name_parent_deriv h (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:

```plaintext
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:

```plaintext
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 \texttt{bus} keyword you define a bus in a derivative definition. Buses are described in section 8.5.2, \textit{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
  - 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.

```plaintext
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.

```plaintext
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:

```latex
\text{group ( ... ) } \\
\{ \\
    \text{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:

```plaintext
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**

```plaintext
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 \texttt{lc\_cb\_section\_name} is defined
as the load-time start address of the section. The symbol
\texttt{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 \texttt{allow\_cross\_references} tells
the linker to accept cross-references. Normally, it does not make sense
to have references between sections that are overlaid.

```c
/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:

```c
group (run_addr = 0xa00f0000)
```

You can use the '``[offset]``' variant to locate the group at the given absolute offset in memory:

```c
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``]

```c
group (run_addr = mem:my_dram)
```

You can use the '``|``' to specify an address range of more than one physical memory device:

```c
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.

```c
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 \texttt{size} you can specify the size for the stack. If the \texttt{size} is not specified, the linker uses the size given by the \texttt{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 \texttt{fixed}.

\begin{verbatim}
 group ( ... )
 {
   stack "mystack" ( size = 2k );
 }
\end{verbatim}

The linker creates two labels to mark the begin and end of the stack, \_\texttt{lc\_ub\_stack\_name} for the begin of the stack and \_\texttt{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 \texttt{stack} keyword in section 8.5.3, \textit{Defining Address Spaces}.

\textbf{Heap}

- The keyword \texttt{heap} tells the linker to reserve a dynamic memory range for the \texttt{malloc()} function. Each heap section has a name. With the keyword \texttt{size} you can change the size for the heap. If the \texttt{size} is not specified, the linker uses the size given by the \texttt{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 \texttt{fixed}.

\begin{verbatim}
 group ( ... )
 {
   heap "myheap" ( size = 2k );
 }
\end{verbatim}

The linker creates two labels to mark the begin and end of the heap, \_\texttt{lc\_ub\_heap\_name} for the begin of the heap and \_\texttt{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.

\textbf{Reserved section}

- The keyword \texttt{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 \texttt{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.

<table>
<thead>
<tr>
<th>Properties set in LSL</th>
<th>Resulting section properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributes</td>
<td>filled</td>
</tr>
<tr>
<td>x</td>
<td>yes</td>
</tr>
<tr>
<td>r</td>
<td>yes</td>
</tr>
<tr>
<td>r</td>
<td>no</td>
</tr>
<tr>
<td>rx</td>
<td>yes</td>
</tr>
<tr>
<td>rw</td>
<td>yes</td>
</tr>
<tr>
<td>rw</td>
<td>no</td>
</tr>
<tr>
<td>rwx</td>
<td>yes</td>
</tr>
</tbody>
</table>

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.

```linker
  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:

```plaintext
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).

```plaintext
group ( ... )
{
    if ( exists( "mysection" ) )
        select "mysection";
    else
        reserved "myreserved" ( size=2k );
}
```
9.1 MISRA-C:1998

This section lists all supported and unsupported MISRA-C:1998 rules.


A number of MISRA-C rules leave room for interpretation. Other rules can only be checked in a limited way. In such cases the implementation decisions and possible restrictions for these rules are listed.

* x means that the rule is not supported by the TASKING C compiler.
(R) is a required rule, (A) is an advisory rule.

1. (R) The code shall conform to standard C, without language extensions

2. (A) Other languages should only be used with an interface standard

3. (A) Inline assembly is only allowed in dedicated C functions

4. (A) Provision should be made for appropriate run-time checking

5. (R) Only use characters and escape sequences defined by ISO C

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’

15. (A) Floating point implementations should comply with a standard

16. (R) The bit representation of floating point numbers shall not be used
   A violation is reported when a pointer to a floating point type is converted to a pointer to an integer type.

17. (R) "typedef" names shall not be reused

18. (A) Numeric constants should be suffixed to indicate type
   A violation is reported when the value of the constant is outside the range indicated by the suffixes, if any.

19. (R) Octal constants (other than zero) shall not be used

20. (R) All object and function identifiers shall be declared before use

21. (R) Identifiers shall not hide identifiers in an outer scope

22. (A) Declarations should be at function scope where possible

23. (A) All declarations at file scope should be static where possible

24. (R) Identifiers shall not have both internal and external linkage

25. (R) Identifiers with external linkage shall have exactly one definition

26. (R) Multiple declarations for objects or functions shall be compatible

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
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)  
#define 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


1.2 (R) No reliance shall be placed on undefined or unspecified behavior.

x 1.3 (R) Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the languages/compilers/assemblers conform.

x 1.4 (R) The compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.

x 1.5 (A) Floating-point implementations should comply with a defined floating-point standard.

Language extensions

2.1 (R) Assembly language shall be encapsulated and isolated.

2.2 (R) Source code shall only use /* . . . */ style comments.
2.3 (R) The character sequence /* shall not be used within a comment.

2.4 (A) Sections of code should not be "commented out".

   In general, it is not possible to decide whether a piece of comment is C code that is commented out, or just some pseudo code. Instead, the following heuristics are used to detect possible C code inside a comment:
   - a line ends with ';', or
   - a line starts with '}', possibly preceded by white space

**Documentation**

x 3.1 (R) All usage of implementation-defined behavior shall be documented.

x 3.2 (R) The character set and the corresponding encoding shall be documented.

x 3.3 (A) The implementation of integer division in the chosen compiler should be determined, documented and taken into account.

3.4 (R) All uses of the #pragma directive shall be documented and explained.

   This rule is really a documentation issue. The compiler will flag all #pragma directives as violations.

3.5 (R) The implementation-defined behavior and packing of bit fields shall be documented if being relied upon.

x 3.6 (R) All libraries used in production code shall be written to comply with the provisions of this document, and shall have been subject to appropriate validation.

**Character sets**

4.1 (R) Only those escape sequences that are defined in the ISO C standard shall be used.

4.2 (R) Trigraphs shall not be used.
Identifiers

5.1 (R) Identifiers (internal and external) shall not rely on the significance of more than 31 characters.

5.2 (R) Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.

5.3 (R) A typedef name shall be a unique identifier.

5.4 (R) A tag name shall be a unique identifier.

5.5 (A) No object or function identifier with static storage duration should be reused.

5.6 (A) No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure and union member names.

5.7 (A) No identifier name should be reused.

Types

6.1 (R) The plain char type shall be used only for storage and use of character values.

6.2 (R) signed and unsigned char type shall be used only for the storage and use of numeric values.

6.3 (A) typedefs that indicate size and signedness should be used in place of the basic types.

6.4 (R) Bit fields shall only be defined to be of type unsigned int or signed int.

6.5 (R) Bit fields of type signed int shall be at least 2 bits long.

Constants

7.1 (R) Octal constants (other than zero) and octal escape sequences shall not be used.

Declarations and definitions

8.1 (R) Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.

8.2 (R) Whenever an object or function is declared or defined, its type shall be explicitly stated.


8.3 (R) For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.

8.4 (R) If objects or functions are declared more than once their types shall be compatible.

8.5 (R) There shall be no definitions of objects or functions in a header file.

8.6 (R) Functions shall be declared at file scope.

8.7 (R) Objects shall be defined at block scope if they are only accessed from within a single function.

8.8 (R) An external object or function shall be declared in one and only one file.

8.9 (R) An identifier with external linkage shall have exactly one external definition.

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 uninitialzied, 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 $\text{const}$ or $\text{volatile}$ 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 $\text{sizeof}$ operator shall not be used on expressions that contain side effects.

12.4 (R) The right-hand operand of a logical $\&\&$ or $\mid\mid$ operator shall not contain side effects.

12.5 (R) The operands of a logical $\&\&$ or $\mid\mid$ shall be primary-expressions.

12.6 (A) The operands of logical operators ($\&\&$, $\mid\mid$ and !) should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to operators other than ($\&\&$, $\mid\mid$ 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.
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

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.
Symbols
#define, 5-13, 5-93
#include, 5-25
#undef, 5-49
__BUILD__, 1-14
__VERSION__, 1-15
_close, 2-22
_Exit, 2-34
_fp_get_exception_mask, 4-8
_fp_get_exception_status, 4-9
_fp_install_trap_handler, 4-9
_fp_set_exception_mask, 4-8
_fp_set_exception_status, 4-9
_fss_break, 2-8
_fss_init, 2-8
_IOFBF, 2-23
_IOLBF, 2-23
_IONBF, 2-23
_lseek, 2-22
_open, 2-22
_read, 2-22
_tolower, 2-5
_unlink, 2-22
_write, 2-22

archiver options
-?, 5-198
-d, 5-199
-f, 5-200
-m, 5-202
-p, 5-203
-r, 5-204
-t, 5-206
-V, 5-207
-w, 5-209
-x, 5-208
add module, 5-204
create library, 5-204
delete module, 5-199
extract module, 5-208
move module, 5-202
print list of objects, 5-206
print list of symbols, 5-206
print module, 5-203
replace module, 5-204
arg, 3-7
ascii, 3-24
asciiz, 3-24
asctime, 2-40
asin functions, 2-11
asinh functions, 2-11
asn, 3-7
aspcp, 3-7
assembler controls
$debug, 3-67
$bw_only, 3-68
$ident, 3-69
$list, 3-72
$list on/off, 3-70
$object, 3-74
$page, 3-75
$prctl, 3-77
$stitle, 3-78
$title, 3-79
$warning off, 3-80
detailed description, 3-66
listing controls (overview), 3-66
miscellaneous (overview), 3-66
overview, 3-66
assembler directives
accum, 3-21
alias, 3-22
align, 3-23
ascii, 3-24
asciiz, 3-24
assembly control (overview), 3-18
byte, 3-25
calls, 3-26
comment, 3-27
conditional assembly (overview), 3-20
data definition (overview), 3-19
debug information (overview), 3-20
define, 3-28
detailed description, 3-20
double, 3-39
dup/endm, 3-29
dupa/endm, 3-30
dupc/endm, 3-31
dupf/endm, 3-32
end, 3-34
equ, 3-35
exitm, 3-36
extern, 3-37
fail, 3-38
float, 3-39
fract, 3-40
global, 3-41
half, 3-64
if, 3-42
include, 3-44
local, 3-45
macro/endm, 3-46
macros (overview), 3-20
message, 3-48
name, 3-49
org, 3-50
overview, 3-18
pmacro, 3-52
sdecl, 3-53
sect, 3-56
set, 3-57
sfract, 3-40
size, 3-58
space, 3-59
storage allocation (overview), 3-19
symbol definitions (overview), 3-19
type, 3-60
undef, 3-61
warning, 3-62
weak, 3-63
word, 3-64
assembler list file, 5-76
assembler options
-?, 5-55
--case-sensitive, 5-57
--check, 5-58
--cpu, 5-56
--debug-info, 5-69
--define, 5-59
--diag, 5-61
--emit-locals, 5-64
--error-file, 5-65
--error-limit, 5-66
--help, 5-55
--include-directory, 5-72
--include-file, 5-71
--keep-output-files, 5-75
--list-file, 5-78
--list-format, 5-76
--no-tasking-sfr, 5-80
--no-warnings, 5-87
--optimize, 5-81
--option-file, 5-67
--output, 5-82
--pcptype, 5-84
--prefix, 5-83
--preprocess, 5-63
--preprocessor-type, 5-79
--section-info, 5-85
--symbol-scope, 5-74
--version, 5-86
--warnings-as-errors, 5-88  flr, 3-10
-C, 5-56           bi, 3-11
-c, 5-57            init_r7, 3-11
-D, 5-59            int, 3-11
-f, 5-67            l10, 3-11
-g, 5-69            len, 3-12
-H, 5-71            lng, 3-12
-I, 5-72            lo, 3-12
-i, 5-74            log, 3-12
-k, 5-75            lsb, 3-12
-L, 5-76            lst, 3-13
-l, 5-78            lun, 3-13
-m, 5-79            mac, 3-13
-O, 5-81            macros (overview), 3-5
-o, 5-82            mathematical (overview), 3-4
-P, 5-83            max, 3-13
-p, 5-84            min, 3-13
-t, 5-85            msb, 3-14
-V, 5-86            mxp, 3-14
-w, 5-87            pos, 3-14
assembly functions
    abs, 3-6           pow, 3-14
    acs, 3-7           rnd, 3-14
address calculation (overview), 3-6    rvb, 3-15
    arg, 3-7           scp, 3-15
    asn, 3-7           sfract, 3-15
aspcp, 3-7           sgn, 3-15
assembler mode (overview), 3-6           sin, 3-15
at2, 3-7           snb, 3-16
atn, 3-8           sqrt, 3-16
fract, 3-8, 3-11    strings (overview), 3-5
    cel, 3-8            sub, 3-16
    cnt, 3-8            syntax, 3-3
    cob, 3-8            tan, 3-16
conversions (overview), 3-5            tnb, 3-16
    cos, 3-9            unf, 3-17
    cpu, 3-9            xpn, 3-17
    cvf, 3-9            at2, 3-7
    cvi, 3-9            atan functions, 2-11
    def, 3-9            atan2 functions, 2-11
    dptr, 3-10          atanh functions, 2-12
    exp, 3-10           atexit, 2-34
    fld, 3-10           atn, 3-8

atoi, 2-32
atol, 2-32
atoll, 2-32

B
binary_switch, 1-12
board specification, 8-5, 8-34
bsearch, 2-34
btowc, 2-44
BUFSIZ, 2-21
bus definition, 8-5
buses, 8-24
byte, 3-25

C
calloc, 2-33
calls, 3-26
case sensitivity, 5-92
cat, 3-8
cbrt functions, 2-15
ceil functions, 2-13
cel, 3-8
char type, treat as unsigned, 5-50
chdir, 2-42
check source code, 5-10, 5-58, 5-136
clear/noclear, 1-10
clearerr, 2-31
clock, 2-40
clock_t, 2-39
CLOCKS_PER_SEC, 2-40
close, 2-42
cnt, 3-8
coh, 3-8
command file, 5-20, 5-67, 5-100,
5-144, 5-184, 5-200
comment, 3-27
comments, 8-7
compact-max-size, 5-11
compactmaxmatch, 1-10

compiler options
-?, 5-4
--align-stack, 5-7
--check, 5-10
--compact-max-size, 5-11
--core, 5-12
--cpu, 5-8
--debug-info, 5-22
--define, 5-13
--diag, 5-15
--error-file, 5-19
--help, 5-4
--include-directory, 5-25
--include-file, 5-24
--inline, 5-27
--inline-max-incr, 5-28
--inline-max-size, 5-28
--iso, 5-9
--keep-output-files, 5-30
--language, 5-5
--misrac, 5-31
--misrac-advisory-warnings, 5-32
--misrac-required-warnings, 5-32
--misrac-version, 5-33
--no-clear, 5-35
--no-stdinc, 5-36
--no-tasking-sfr, 5-37, 5-38
--no-warnings, 5-52
--option-file, 5-20
--output, 5-42
--preprocess, 5-17
--rename-sections, 5-43
--signed-bitfields, 5-46
--source, 5-45
--static, 5-47
--stdout, 5-34
--tradeoff, 5-48
--uchar, 5-50
--undefined, 5-49
--version, 5-51
--warnings-as-errors, 5-53
-A, 5-5
-C, 5-8
-c, 5-9
-D, 5-13
-E, 5-17
-f, 5-20
-g, 5-22
-H, 5-24
-I, 5-25
-k, 5-30
-n, 5-34
--optimize, 5-39
-O, 5-39
-o, 5-42
-R, 5-43
-s, 5-45
-t, 5-48
-U, 5-49
-u, 5-50
-V, 5-51
-w, 5-52
conditional make rules, 5-171
control program options
-?, 5-131
--address-size, 5-132
--case-sensitive, 5-134
--check, 5-136
--cpu, 5-133
--create, 5-135
--d, 5-139
--debug-info, 5-148
--define, 5-137
--diag, 5-140
--dry-run, 5-156
--error-file, 5-142
--format, 5-146
--fptrap, 5-147
--hlp, 5-131
--ignore-default-library-path, 5-152
--include-directory, 5-149
--iso, 5-150
--keep-output-files, 5-151
--library, 5-154
--library-directory, 5-152
--list-files, 5-155
--lsl-file, 5-139
--no-default-libraries, 5-157
--no-double, 5-143
--no-map-file, 5-158
--no-tasking-sfr, 5-159
--no-warnings, 5-168
--option-file, 5-144
--output, 5-160
--pass, 5-167
--pass-assembler, 5-167
--pass-c, 5-167
--pass-linker, 5-167
--preprocess, 5-141
--space, 5-161
--static, 5-162
--undefined, 5-164
--verbose, 5-166
--version, 5-165
--warnings-as-errors, 5-169
-C, 5-133
-cl, 5-135
-co, 5-135
-cs, 5-135
-D, 5-137
-E, 5-141
-F, 5-143
-f, 5-144
-g, 5-148
-I, 5-149
-k, 5-151
-L, 5-152
-l, 5-154
-n, 5-156
-o, 5-160
-t, 5-163
-U, 5-164
-V, 5-165
-v, 5-166
-W, 5-167
-w, 5-168
-Wa, 5-167
controls

See also assembler directives
detailed description, 3-66

copy table, 5-117, 8-26, 8-51
copysign functions, 2-15
core type, 5-56
cos, 3-9
cos functions, 2-11
cosh functions, 2-11
cpu, 3-9
CPU type, 5-8, 5-56, 5-133
cstart.asm, 4-3
cftime, 2-40
cvf, 3-9
cvi, 3-9
cycle count, 5-85

data types, 1-4
debug, 3-67
debug information, 5-22, 5-69, 5-125
def, 3-9
define, 3-28
derivative definition, 8-4, 8-30
difftime, 2-40
directives
See also assembler directives
detailed description, 3-20
div, 2-35
double, 3-39
dptr, 3-10
dup, 3-29
dupa, 3-30
dupc, 3-31
dupf, 3-32

ELF/DWARF object format, 7-3
elif, 3-42
eelse, 3-42
dend, 3-34
dendif, 3-42
EOF, 2-21
equ, 3-35
erf functions, 2-16
erfc functions, 2-16
erro, 2-5
exceptions, floating-point, 4-6
exit, 2-34
exit macro, 3-36
EXIT_FAILURE, 2-32
EXIT_SUCCESS, 2-32
exitm, 3-36
exp, 3-10
exp functions, 2-12
exp2 functions, 2-12
expm1 functions, 2-12
extension is suffix, 1-11
extern, 1-11, 3-37

fabs functions, 2-15
fail, 3-38
fclose, 2-22
fdim functions, 2-16
FE_ALL_EXCEPT, 2-8
FE_DIVBYZERO, 2-8
FE_INEXACT, 2-8
FE_INVALID, 2-8
FE_OVERFLOW, 2-8
FE_UNDERFLOW, 2-8
Index

feclearexcept, 2-7
fegetenv, 2-7
fegetexceptflag, 2-7
feholdexcept, 2-7
ferror, 2-31
fesetenv, 2-7
fesetexceptflag, 2-7
fetestexcept, 2-7, 2-8
fesetenv, 2-7
fflush, 2-23
fgetc, 2-27
fgetpos, 2-30
fgets, 2-27
fgetwc, 2-27
fgets, 2-27
fflush, 2-23
ferror, 2-31
fesetenv, 2-7
fesetexceptflag, 2-7
fetestexcept, 2-7, 2-8
fesetenv, 2-7
fflush, 2-23
fgetwc, 2-27
fgets, 2-27
fgetwc, 2-27
filesystem simulation, 2-4
FILENAME_MAX, 2-21
fld, 3-10
float, 3-39
floating-point, 4-4
libraries, 4-6
special values, 4-6
trap handler, 4-7
trap handling api, 4-8
trapping, 4-6
floor functions, 2-13
flr, 3-10
fma functions, 2-15
fmax functions, 2-16
fmin functions, 2-16
fmod functions, 2-14
fopen, 2-22
FOPEN_MAX, 2-21
fpclassify, 2-17
fprintf, 2-29
fputc, 2-28
fputs, 2-28
fputwc, 2-28
fputws, 2-28
fread, 2-30
free, 2-33
freopen, 2-23
frexp functions, 2-14
fscanf, 2-27
fseek, 2-30
fsetpos, 2-30
FSS, 2-4
ftell, 2-30
functions, assembly, 3-3
fwprintf, 2-29
fwrite, 2-30
fwscanf, 2-27

G
get, 2-27
getchar, 2-27
getcwd, 2-42
getenv, 2-34
gets, 2-27
getwc, 2-27
getwchar, 2-27
global, 3-41
gmtime, 2-40

H
half, 3-64
Header files, 2-4
alert.b, 2-4
ctype.b, 2-4
ero.b, 2-5
fcntl.b, 2-7
fenv.b, 2-7
float.b, 2-8
fss.b, 2-8
inttypes.b, 2-9
iso646.b, 2-10
limits.b, 2-10
locale.b, 2-10
math.h, 2-11
setjmp.h, 2-18
signal.h, 2-18
stdarg.h, 2-19
stdbool.h, 2-19
stddef.h, 2-20
stdint.h, 2-9
stdio.h, 2-20
stdlib.h, 2-31
string.h, 2-35
tgmath.h, 2-11
time.h, 2-39
unistd.h, 2-42
wchar.h, 2-20, 2-35, 2-39, 2-43
wctype.h, 2-4, 2-44
heap, 4-4, 8-25
begin of, 4-4
dead of, 4-4
hi, 3-11
hw_only, 3-68
hypot functions, 2-15

ident, 3-69
if, 3-42
ilogb functions, 2-12
imaxabs, 2-9
imaxdiv, 2-9
include, 3-44
init_r7, 3-11
inline functions, 5-28
inline/noinline, 1-11
int, 3-11
Intel hex, record type, 7-8
intrinsic functions, 1-8
__alloc(), 1-8
__get_return_address(), 1-8
__ld32_fpi(), 1-8
__nop(), 1-8
__st32_fpi(), 1-9
isalnum, 2-4
isalpha, 2-4
isblank, 2-4
iscntrl, 2-4
isdigit, 2-4
isfinite, 2-17
isgraph, 2-4
isgreater, 2-17
isgreaterequal, 2-17
isnan, 2-17
isinf, 2-17
isless, 2-17
islessequal, 2-17
islessequal, 2-17
islower, 2-4
isnormal, 2-17
ISO C standard, 5-9
isprint, 2-5
ispunct, 2-5
isspace, 2-5
isunordered, 2-17
isupper, 2-5
iswalnum, 2-4, 2-44
iswalpha, 2-4, 2-44
iswblank, 2-4
iswcntrl, 2-4, 2-44
iswctype, 2-44
iswdigit, 2-4, 2-44
iswgraph, 2-4, 2-44
iswlower, 2-4, 2-45
iswprint, 2-5, 2-45
iswpunct, 2-5, 2-45
iswspace, 2-5, 2-45
iswupper, 2-5, 2-45
iswxdigit, 2-5
<table>
<thead>
<tr>
<th>Subject</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>iswdx ditig</td>
<td>2-45</td>
</tr>
<tr>
<td>isxdigit</td>
<td>2-5</td>
</tr>
<tr>
<td>J</td>
<td></td>
</tr>
<tr>
<td>jump switch</td>
<td>1-12</td>
</tr>
<tr>
<td>L</td>
<td></td>
</tr>
<tr>
<td>L_tmpnam</td>
<td>2-21</td>
</tr>
<tr>
<td>l10</td>
<td>3-11</td>
</tr>
<tr>
<td>labs</td>
<td>2-35</td>
</tr>
<tr>
<td>language extensions, intrinsic</td>
<td></td>
</tr>
<tr>
<td>functions</td>
<td>1-8</td>
</tr>
<tr>
<td>ldexp functions</td>
<td>2-14</td>
</tr>
<tr>
<td>ldiv</td>
<td>2-35</td>
</tr>
<tr>
<td>len</td>
<td>3-12</td>
</tr>
<tr>
<td>lgamma functions</td>
<td>2-16</td>
</tr>
<tr>
<td>linear_switch</td>
<td>1-12</td>
</tr>
<tr>
<td>linker map file</td>
<td>5-112</td>
</tr>
<tr>
<td>linker options</td>
<td></td>
</tr>
<tr>
<td>-?, -?</td>
<td>5-90</td>
</tr>
<tr>
<td>--case-insensitive</td>
<td>5-92</td>
</tr>
<tr>
<td>--chip-output</td>
<td>5-91</td>
</tr>
<tr>
<td>--define</td>
<td>5-93</td>
</tr>
<tr>
<td>--diag</td>
<td>5-95</td>
</tr>
<tr>
<td>--error-file</td>
<td>5-98</td>
</tr>
<tr>
<td>--error-limit</td>
<td>5-99</td>
</tr>
<tr>
<td>--extern</td>
<td>5-97</td>
</tr>
<tr>
<td>--extra-verbose</td>
<td>5-127</td>
</tr>
<tr>
<td>--first-library-first</td>
<td>5-102</td>
</tr>
<tr>
<td>--help</td>
<td>5-90</td>
</tr>
<tr>
<td>--ignore-default-library-path</td>
<td>5-106</td>
</tr>
<tr>
<td>--include-directory</td>
<td>5-103</td>
</tr>
<tr>
<td>--incremental</td>
<td>5-124</td>
</tr>
<tr>
<td>--keep-output-files</td>
<td>5-105</td>
</tr>
<tr>
<td>--library</td>
<td>5-108</td>
</tr>
<tr>
<td>--library-directory</td>
<td>5-106</td>
</tr>
<tr>
<td>--link-only</td>
<td>5-109</td>
</tr>
<tr>
<td>--lsl-check</td>
<td>5-110</td>
</tr>
<tr>
<td>--lsl-dump</td>
<td>5-111</td>
</tr>
<tr>
<td>--map-file</td>
<td>5-112</td>
</tr>
<tr>
<td>--map-file-format</td>
<td>5-113</td>
</tr>
<tr>
<td>--misra-c-report</td>
<td>5-115</td>
</tr>
<tr>
<td>--munch</td>
<td>5-116</td>
</tr>
<tr>
<td>--no-rescan</td>
<td>5-118</td>
</tr>
<tr>
<td>--no-rom-copy</td>
<td>5-117</td>
</tr>
<tr>
<td>--no-warnings</td>
<td>5-128</td>
</tr>
<tr>
<td>--non-romable</td>
<td>5-119</td>
</tr>
<tr>
<td>--optimize</td>
<td>5-120</td>
</tr>
<tr>
<td>--option-file</td>
<td>5-100</td>
</tr>
<tr>
<td>--output</td>
<td>5-122</td>
</tr>
<tr>
<td>--strip-debug</td>
<td>5-125</td>
</tr>
<tr>
<td>--user-provided-initialization-code</td>
<td>5-104</td>
</tr>
<tr>
<td>--verbose</td>
<td>5-127</td>
</tr>
<tr>
<td>--version</td>
<td>5-126</td>
</tr>
<tr>
<td>--warnings-as-errors</td>
<td>5-129</td>
</tr>
<tr>
<td>-c</td>
<td>5-91</td>
</tr>
<tr>
<td>-D</td>
<td>5-93</td>
</tr>
<tr>
<td>-d</td>
<td>5-94</td>
</tr>
<tr>
<td>-e</td>
<td>5-97</td>
</tr>
<tr>
<td>-f</td>
<td>5-100</td>
</tr>
<tr>
<td>-I</td>
<td>5-103</td>
</tr>
<tr>
<td>-i</td>
<td>5-104</td>
</tr>
<tr>
<td>-k</td>
<td>5-105</td>
</tr>
<tr>
<td>-L</td>
<td>5-106</td>
</tr>
<tr>
<td>-l</td>
<td>5-108</td>
</tr>
<tr>
<td>-M</td>
<td>5-112</td>
</tr>
<tr>
<td>-m</td>
<td>5-113</td>
</tr>
<tr>
<td>-N</td>
<td>5-117</td>
</tr>
<tr>
<td>-O</td>
<td>5-120</td>
</tr>
<tr>
<td>-o</td>
<td>5-122</td>
</tr>
<tr>
<td>-r</td>
<td>5-124</td>
</tr>
<tr>
<td>-S</td>
<td>5-125</td>
</tr>
<tr>
<td>-V</td>
<td>5-126</td>
</tr>
<tr>
<td>-v</td>
<td>5-127</td>
</tr>
<tr>
<td>-vv</td>
<td>5-127</td>
</tr>
<tr>
<td>-w</td>
<td>5-128</td>
</tr>
<tr>
<td>linker script file</td>
<td></td>
</tr>
<tr>
<td>architecture definition</td>
<td>8-3</td>
</tr>
<tr>
<td>board specification</td>
<td>8-5</td>
</tr>
<tr>
<td>bus definition</td>
<td>8-5</td>
</tr>
</tbody>
</table>
derivative definition, 8-4
memory definition, 8-5
preprocessing, 8-6
processor definition, 8-4
section layout definition, 8-5
structure, 8-3
list, 3-72
list file, 5-78
 assembler, 5-76
 linker, 5-112
list on/off, 3-70
llabs, 2-35
ldiv, 2-35
llrint functions, 2-13
llround functions, 2-13
lng, 3-12
lo, 3-12
local, 3-45
localeconv, 2-11
localtime, 2-40
log, 3-12
log functions, 2-12
log10 functions, 2-12
log1p functions, 2-12
log2 functions, 2-12
logb functions, 2-12
longjmp, 2-18
lrint functions, 2-13
lround functions, 2-13
lsb, 3-12
lseek, 2-42
LSL expression evaluation, 8-21
LSL functions
 absolute(), 8-9
 addressof(), 8-9
 exists(), 8-10
 max(), 8-10
 min(), 8-10
 sizeof(), 8-10
LSL keywords
 align, 8-24, 8-25, 8-26, 8-42
 alloc_allowed, 8-48
 allow_cross_references, 8-44
architecture, 8-23, 8-31
attributes, 8-41, 8-42
blocks, 8-50
bus, 8-24, 8-27, 8-36
clustered, 8-43
contiguous, 8-43
copy_unit, 8-26
copytable, 8-26, 8-51
core, 8-31
derivative, 8-30, 8-35
dest, 8-26, 8-27
dest_dbits, 8-28
dest_offset, 8-28
direction, 8-25, 8-39, 8-43
dust, 8-52
extends, 8-23, 8-30
fill, 8-43, 8-48, 8-49
fixed, 8-25, 8-47
group, 8-40, 8-42
grows, 8-25
heap, 8-25, 8-47
high_to_low, 8-25, 8-39
id, 8-24
if, 8-52
load_addr, 8-45
low_to_high, 8-25, 8-39
map, 8-24, 8-25, 8-27, 8-32
mau, 8-24, 8-32, 8-36
mem, 8-45
memory, 8-32, 8-36
min_size, 8-25, 8-47
nvram, 8-32
ordered, 8-43
overflow, 8-49
overlay, 8-43
page, 8-26, 8-46
page_size, 8-25, 8-46
processor, 8-34
ram, 8-32
ref_tree, 8-41
reserved, 8-32, 8-47
rom, 8-32
run_addr, 8-26, 8-45
Index

section, 8-49
section_layout, 8-39
select, 8-40
size, 8-28, 8-32, 8-36, 8-47, 8-49
space, 8-24, 8-27
speed, 8-32, 8-36
src_dbits, 8-28
src_offset, 8-28
stack, 8-25, 8-46
start_address, 8-26
symbol, 8-27
type, 8-32, 8-36
width, 8-24
LSL syntax, 8-6
lst, 3-13
lun, 3-13

M

mac, 3-13
macro, 3-46
define, 5-137
definition, 3-46
undefine, 3-52, 5-164
macro/nomacro, 1-11
macros, 1-14
make utility, 5-171
macros, predefined
_DATE_, 5-49
_FILE_, 5-49
_LINE_, 5-49
_STDINC_, 5-49
_TIME_, 5-49
make utility options
-?, 5-173
-a, 5-174
-D, 5-175
-d, 5-176
-DD, 5-175
-dd, 5-176
-e, 5-177
-err, 5-178
-f, 5-179
-G, 5-180
-i, 5-181
-K, 5-182
-k, 5-183
-m, 5-184, 5-190
-n, 5-186
-p, 5-187
-q, 5-188
-r, 5-189
-s, 5-191
-t, 5-192
-time, 5-193
-V, 5-194
-W, 5-195
-w, 5-196
defining a macro, 5-171
malloc, 2-33
map file
control program option, 5-158
format, 5-113
linker, 5-112
mappings, 8-27
max, 3-13
maxcallddepth, 1-11
MB_CUR_MAX, 2-32, 2-43
MB_LEN_MAX, 2-43
mblen, 2-35
mbrlen, 2-44
mbtowc, 2-43
mbsinit, 2-43
mbsrtowcs, 2-43
mbstate_t, 2-43
mbstowcs, 2-35
mbtowc, 2-35
memchr, 2-38
memcmp, 2-37
memcpy, 2-36
memmove, 2-36
memory definition, 8-5
memset, 2-39
INDEX

message, 1-11, 3-48
min, 3-13
MISRA-C, 5-31, 5-32

supported rules 1998, 9-3

supported rules 2004, 9-10

version, 5-33

mtime, 2-40
mod function, 2-14
msb, 3-14
mxp, 3-14

N

name, 3-49

nan functions, 2-15

nearbyint functions, 2-13

nextafter functions, 2-15

nexttoward functions, 2-15

novector, 1-12

NULL, 2-20

O

object, 3-74

offsetof, 2-20

open, 2-7

optimization, 5-39, 5-81, 5-120

optimize/endoptimize, 1-12

option file, 5-20, 5-67, 5-100, 5-144, 5-184, 5-200

org, 3-50

output file, 5-42, 5-82, 5-122, 5-160

output format, 5-91, 5-146

P

page, 3-75

pass option to tool, 5-167

PCP syntax, 5-84

perror, 2-31

pmacro, 3-52

pos, 3-14

pow, 3-14

pow functions, 2-15

Pragma

extern, 1-11

macro, 1-11

message, 1-11

protect, 1-12

tradeoff, 1-13

warning, 1-13

weak, 1-13

Pragmas

alias, 1-10

clear/noclear, 1-10

compactmaxmatch, 1-10

extension is suffix, 1-11

inline/noinline, 1-11

maxcalldepth, 1-11

novector, 1-12

optimize/endoptimize, 1-12

section, 1-12

smartinline, 1-11

source/nosource, 1-12

stinct, 1-12

pragmas, 1-10

prctl, 3-77

predefined macros, 1-14

predefined macros in C

__BIGENDIAN__, 1-14

__CORE__, 1-14

__CPCP__, 1-14

__DATE__, 1-14

__DOUBLE_FP__, 1-14

__FILE__, 1-14

__LINE__, 1-14

__REVISION__, 1-14

__SFRFILE__, 1-15

__SINGLE_FP__, 1-15

__STDC__, 1-15

__STDC_HOSTED__, 1-15

__STDC_VERSION__, 1-15
__TASKING__, 1-15
__TIME__, 1-15
preprocessing, 8-6
preprocessor, 5-79
printf, 2-23, 2-29
conversion characters, 2-25
processor definition, 8-4, 8-34
protect, 1-12
ptrdiff_t, 2-20
putc, 2-28
putchar, 2-28
puts, 2-29
putwc, 2-28
putwchar, 2-28
qsort, 2-34
raise, 2-18
rand, 2-33
RAND_MAX, 2-32
read, 2-42
realloc, 2-33
remainder functions, 2-14
remove, 2-31
remquo functions, 2-14
rename, 2-31
rename sections, 5-43
reset vector, 8-26
rewind, 2-30
rint functions, 2-13
rnd, 3-14
round functions, 2-13
rvb, 3-15
scalbln functions, 2-14
scalbn functions, 2-14
scanf, 2-25, 2-28
summary, 5-85
section activation, 3-56
section attributes, 3-53
section declaration, 3-53
section layout definition, 8-5, 8-38
section names, 3-55
setbuf, 2-23
setjmp, 2-18
setlocale, 2-10
setvbuf, 2-23
sfract, 3-15, 3-40
sgn, 3-15
SIGABRT, 2-18
SIGFPE, 2-18
SIGFPE signal handler, 4-7
SIGILL, 2-18
SIGINT, 2-18
signal, 2-18
signbit, 2-17
SIGSEGV, 2-18
SIGTERM, 2-18
sin, 3-15
sin functions, 2-11
sinh functions, 2-11
size, 3-58
size_t, 2-20
smart_switch, 1-12
smartinline, 1-11
snh, 3-16
snprintf, 2-29
source/nosource, 1-12
space, 3-59
sprintf, 2-29
sqrt functions, 2-15
sqt, 3-16
srand, 2-33
sscanf, 2-28
stack, 4-3, 8-25
start address, 8-26
startup code, 4-3
stat, 2-42
static stack alignment, 5-7
stderr, 2-21
stdin, 2-21
stdinc, 1-12
stdout, 2-21
stitle, 3-78
strcat, 2-36
strchr, 2-38
strcmp, 2-37
strcoll, 2-37
strncpy, 2-36
strcspn, 2-38
strerror, 2-39
strncat, 2-36
strncmp, 2-37
strncpy, 2-36
strpbrk, 2-38
strrchr, 2-38
strspn, 2-38
strstr, 2-38
strtod, 2-32
strtof, 2-32
strtoimax, 2-9
strtok, 2-38
strtol, 2-33
strtold, 2-32
strtoll, 2-33
strtol, 2-33
strtoimax, 2-9
strxfrm, 2-37
sub, 3-16
Switch method, 1-12
swprintf, 2-29
swscanf, 2-28
syntax error checking, 5-10, 5-12,
5-58, 5-136
system, 2-34
system libraries, 5-106, 5-108

tan, 3-16
tan functions, 2-11
tanh functions, 2-11
temporary files, 5-163
tgamma functions, 2-16
time, 2-40
time_t, 2-39
tm (struct), 2-39
TMP_MAX, 2-21
tmpfile, 2-31
tmpnam, 2-31
tnh, 3-16
tolower, 2-5
toupper, 2-5
towctrans, 2-45
towlower, 2-5, 2-45
towupper, 2-5, 2-45
tradeoff, 1-13
trap, 4-11
trap handler, 4-7
trap handling, 5-147
trap handling api, 4-8
trunc functions, 2-13

U

undef, 3-61
unf, 3-17
ungetc, 2-27
ungetwc, 2-27
unlink, 2-42

V

va_arg, 2-19
va_end, 2-19
va_start, 2-19
verbose, 5-127, 5-166
version information, 5-51, 5-86, 5-126,
5-165, 5-194, 5-195, 5-207
vfprintf, 2-29
vfscanf, 2-28
vfprintf, 2-29
vfscanf, 2-28
vfwprintf, 2-29
vfwsprintf, 2-29
vprintf, 2-29
scanf, 2-28
vscanf, 2-28
vsprintf, 2-29
vsscanf, 2-28
vswprintf, 2-29
vswscanf, 2-28
vwprintf, 2-29
vwscanf, 2-28

W

warning, 1-13, 3-62
title, 3-79, 3-80
warnings, 5-169
  suppress, 5-87
warnings as errors, 5-53, 5-88, 5-129
warnings, suppress, 5-52, 5-128
wchar_t, 2-20
wcrtomb, 2-44
wcsat, 2-36
wcschr, 2-38
wcsncmp, 2-37
wcscoll, 2-37
wcsncmp, 2-37
wcsncat, 2-36
wcsncmp, 2-37
wcsncpy, 2-36
wcsrchr, 2-38
wcsrtombs, 2-43
wcspn, 2-38
wcsrt, 2-38
wcestod, 2-32
wcstof, 2-32
wcstoi, 2-32
wcstoul, 2-33
wcstoul, 2-33
wcstoull, 2-33
wcstoumax, 2-9
wcsxfrm, 2-37
wctob, 2-44
wcstomb, 2-35
wcstrans, 2-45
wctype, 2-44
weak, 1-13, 3-63
WEOF, 2-21
wmemchr, 2-38
wmemcmp, 2-37
wmemcpy, 2-36
wmemmove, 2-36
wmemset, 2-39
word, 3-64
wprintf, 2-29
write, 2-42
wscanf, 2-28
wstrftime, 2-41

X

xpn, 3-17