



TASKING VX-toolset for PCP User Guide

TASKING VX-toolset for PCP User Guide

Copyright © 2010 Altium Limited.

All rights reserved. You are permitted to print this document provided that (1) the use of such is for personal use only and will not be copied or posted on any network computer or broadcast in any media, and (2) no modifications of the document is made. Unauthorized duplication, in whole or part, of this document by any means, mechanical or electronic, including translation into another language, except for brief excerpts in published reviews, is prohibited without the express written permission of Altium Limited. Unauthorized duplication of this work may also be prohibited by local statute. Violators may be subject to both criminal and civil penalties, including fines and/or imprisonment. Altium, TASKING, and their respective logos are trademarks or registered trademarks of Altium Limited or its subsidiaries. All other registered or unregistered trademarks referenced herein are the property of their respective owners and no trademark rights to the same are claimed.

Table of Contents

1. C Language	1
1.1. Data Types	1
1.1.1. Changing the Alignment: <code>__align()</code>	2
1.2. Accessing Memory	3
1.2.1. Memory Type Qualifiers	3
1.2.2. Pointers	4
1.2.3. Placing an Object at an Absolute Address: <code>__at()</code>	5
1.2.4. Accessing Hardware from C	6
1.3. Using Assembly in the C Source: <code>__asm()</code>	7
1.4. Attributes	11
1.5. Pragmas to Control the Compiler	15
1.6. Predefined Preprocessor Macros	18
1.7. Switch Statement	20
1.8. Functions	21
1.8.1. Calling Convention	21
1.8.2. Register Usage	22
1.8.3. Inlining Functions: <code>inline</code>	22
1.8.4. Interrupt Functions	24
1.8.5. Intrinsic Functions	26
1.9. PCP Code Generation	33
1.9.1. Non-interruptible Code Generation	33
1.9.2. Interruptible Code Generation	34
1.10. Compiler Generated Sections	36
1.10.1. Rename Sections	37
2. Assembly Language	39
2.1. Assembly Syntax	39
2.2. Assembler Significant Characters	40
2.3. Operands of an Assembly Instruction	40
2.4. Symbol Names	41
2.4.1. Predefined Preprocessor Symbols	42
2.5. Registers	42
2.5.1. Special Function Registers	42
2.6. Assembly Expressions	43
2.6.1. Numeric Constants	43
2.6.2. Strings	44
2.6.3. Expression Operators	45
2.7. Working with Sections	46
2.8. Built-in Assembly Functions	47
2.9. Assembler Directives and Controls	59
2.9.1. Assembler Directives	60
2.9.2. Assembler Controls	107
2.10. Macro Operations	121
2.10.1. Defining a Macro	121
2.10.2. Calling a Macro	121
2.10.3. Using Operators for Macro Arguments	122
2.11. Generic Instructions	126
3. Using the C Compiler	129
3.1. Compilation Process	129

3.2. Calling the C Compiler	130
3.3. The C Startup Code	132
3.4. How the Compiler Searches Include Files	132
3.5. Compiling for Debugging	133
3.6. Compiler Optimizations	134
3.6.1. Generic Optimizations (frontend)	135
3.6.2. Core Specific Optimizations (backend)	138
3.6.3. Optimize for Size or Speed	139
3.6.4. Static Stack Alignment Optimizations	142
3.7. Static Code Analysis	142
3.7.1. C Code Checking: CERT C	144
3.7.2. C Code Checking: MISRA-C	145
3.8. C Compiler Error Messages	147
4. Using the Assembler	149
4.1. Assembly Process	149
4.2. Calling the Assembler	150
4.3. How the Assembler Searches Include Files	151
4.4. Assembler Optimizations	152
4.5. Generating a List File	152
4.6. Assembler Error Messages	153
5. Using the Linker	155
5.1. Linking Process	155
5.1.1. Phase 1: Linking	157
5.1.2. Phase 2: Locating	158
5.2. Calling the Linker	159
5.3. Linking with Libraries	160
5.3.1. How the Linker Searches Libraries	162
5.3.2. How the Linker Extracts Objects from Libraries	162
5.4. Incremental Linking	163
5.5. Importing Binary Files	164
5.6. Linker Optimizations	164
5.7. Controlling the Linker with a Script	165
5.7.1. Purpose of the Linker Script Language	166
5.7.2. Eclipse and LSL	166
5.7.3. Structure of a Linker Script File	168
5.7.4. The Architecture Definition	171
5.7.5. The Derivative Definition	174
5.7.6. The Processor Definition	176
5.7.7. The Memory Definition	176
5.7.8. The Section Layout Definition: Locating Sections	178
5.8. Linker Labels	180
5.9. Generating a Map File	183
5.10. Linker Error Messages	184
6. Using the Utilities	187
6.1. Control Program	187
6.2. Make Utility mkpcp	188
6.2.1. Calling the Make Utility	190
6.2.2. Writing a Makefile	190
6.3. Make Utility amk	199
6.3.1. Makefile Rules	199

6.3.2. Makefile Directives	201
6.3.3. Macro Definitions	201
6.3.4. Makefile Functions	202
6.3.5. Conditional Processing	203
6.3.6. Makefile Parsing	204
6.3.7. Makefile Command Processing	204
6.3.8. Calling the amk Make Utility	205
6.4. Archiver	206
6.4.1. Calling the Archiver	206
6.4.2. Archiver Examples	208
7. Using the Debugger	211
7.1. Reading the Eclipse Documentation	211
7.2. Creating a Customized Debug Configuration	211
7.3. Troubleshooting	217
7.4. TASKING Debug Perspective	218
7.4.1. Debug View	219
7.4.2. Breakpoints View	221
7.4.3. File System Simulation (FSS) View	222
7.4.4. Disassembly View	223
7.4.5. Expressions View	223
7.4.6. Memory View	224
7.4.7. Compare Application View	225
7.4.8. Heap View	225
7.4.9. Logging View	225
7.4.10. RTOS View	225
7.4.11. TASKING Registers View	226
7.4.12. Trace View	227
7.5. PCP Simulator Configuration	227
8. Tool Options	229
8.1. C Compiler Options	233
8.2. Assembler Options	294
8.3. Linker Options	331
8.4. Control Program Options	377
8.5. Make Utility Options	422
8.6. Parallel Make Utility Options	450
8.7. Archiver Options	460
9. Libraries	475
9.1. Library Functions	475
9.1.1. assert.h	475
9.1.2. complex.h	475
9.1.3. cstart.h	477
9.1.4. ctype.h and wctype.h	477
9.1.5. dbg.h	478
9.1.6. errno.h	478
9.1.7. fcntl.h	479
9.1.8. fenv.h	479
9.1.9. float.h	480
9.1.10. inttypes.h and stdint.h	480
9.1.11. io.h	481
9.1.12. iso646.h	481

9.1.13. limits.h	482
9.1.14. locale.h	482
9.1.15. malloc.h	482
9.1.16. math.h and tgmth.h	483
9.1.17. setjmp.h	487
9.1.18. signal.h	487
9.1.19. stdarg.h	488
9.1.20. stdbool.h	488
9.1.21. stddef.h	489
9.1.22. stdint.h	489
9.1.23. stdio.h and wchar.h	489
9.1.24. stdlib.h and wchar.h	496
9.1.25. string.h and wchar.h	499
9.1.26. time.h and wchar.h	501
9.1.27. unistd.h	504
9.1.28. wchar.h	504
9.1.29. wctype.h	505
9.2. C Library Reentrancy	506
10. List File Formats	519
10.1. Assembler List File Format	519
10.2. Linker Map File Format	520
11. Linker Script Language (LSL)	525
11.1. Structure of a Linker Script File	525
11.2. Syntax of the Linker Script Language	527
11.2.1. Preprocessing	527
11.2.2. Lexical Syntax	528
11.2.3. Identifiers and Tags	528
11.2.4. Expressions	529
11.2.5. Built-in Functions	529
11.2.6. LSL Definitions in the Linker Script File	531
11.2.7. Memory and Bus Definitions	531
11.2.8. Architecture Definition	533
11.2.9. Derivative Definition	536
11.2.10. Processor Definition and Board Specification	537
11.2.11. Section Layout Definition and Section Setup	537
11.3. Expression Evaluation	542
11.4. Semantics of the Architecture Definition	542
11.4.1. Defining an Architecture	543
11.4.2. Defining Internal Buses	544
11.4.3. Defining Address Spaces	544
11.4.4. Mappings	548
11.5. Semantics of the Derivative Definition	550
11.5.1. Defining a Derivative	550
11.5.2. Instantiating Core Architectures	551
11.5.3. Defining Internal Memory and Buses	552
11.6. Semantics of the Board Specification	553
11.6.1. Defining a Processor	553
11.6.2. Instantiating Derivatives	554
11.6.3. Defining External Memory and Buses	554
11.7. Semantics of the Section Setup Definition	555

11.7.1. Setting up a Section	556
11.8. Semantics of the Section Layout Definition	556
11.8.1. Defining a Section Layout	557
11.8.2. Creating and Locating Groups of Sections	558
11.8.3. Creating or Modifying Special Sections	563
11.8.4. Creating Symbols	567
11.8.5. Conditional Group Statements	567
12. Debug Target Configuration Files	569
12.1. Custom Board Support	569
12.2. Description of DTC Elements and Attributes	570
12.3. Special Resource Identifiers	572
12.4. Initialize Elements	573
13. CPU Problem Bypasses and Checks	575
14. CERT C Secure Coding Standard	579
14.1. Preprocessor (PRE)	579
14.2. Declarations and Initialization (DCL)	580
14.3. Expressions (EXP)	581
14.4. Integers (INT)	582
14.5. Floating Point (FLP)	582
14.6. Arrays (ARR)	582
14.7. Characters and Strings (STR)	583
14.8. Memory Management (MEM)	583
14.9. Environment (ENV)	584
14.10. Signals (SIG)	584
14.11. Miscellaneous (MSC)	584
15. MISRA-C Rules	585
15.1. MISRA-C:1998	585
15.2. MISRA-C:2004	589

Chapter 1. C Language

This chapter describes the target specific features of the C language, including language extensions that are not standard in ISO-C. For example, pragmas are a way to control the compiler from within the C source.

The TASKING C compiler(s) fully support the ISO-C standard and add extra possibilities to program the special functions of the target.

In addition to the standard C language, the compiler supports the following:

- keywords to specify memory types for data and functions
- attribute to specify alignment and absolute addresses
- intrinsic (built-in) functions that result in target specific assembly instructions
- pragmas to control the compiler from within the C source
- predefined macros
- the possibility to use assembly instructions in the C source
- keywords for inlining functions and programming interrupt routines
- libraries

All non-standard keywords have two leading underscores (__).

In this chapter the target specific characteristics of the C language are described, including the above mentioned extensions.

1.1. Data Types

The C compiler supports the ISO C99 defined data types. The sizes of these types are shown in the following table.

C Type	Size	Align	Limits
_Bool	32	32	0 or 1
__far __mau8 signed char *	8	8	$[-2^7, 2^7-1]$
signed char	32	32	$[-2^{31}, 2^{31}-1]$
__far __mau8 unsigned char *	8	8	$[0, 2^8-1]$
unsigned char	32	32	$[0, 2^{32}-1]$
__far __mau8 short *	16	16	$[-2^{15}, 2^{15}-1]$
short	32	32	$[-2^{31}, 2^{31}-1]$
__far __mau8 unsigned short *	16	16	$[0, 2^{16}-1]$
unsigned short	32	32	$[0, 2^{32}-1]$
int	32	32	$[-2^{31}, 2^{31}-1]$

C Type	Size	Align	Limits
unsigned int	32	32	$[0, 2^{32}-1]$
enum	32	32	$[-2^{31}, 2^{31}-1]$
long	32	32	$[-2^{31}, 2^{31}-1]$
unsigned long	32	32	$[0, 2^{32}-1]$
long long	32	32	$[-2^{31}, 2^{31}-1]$
unsigned long long	32	32	$[0, 2^{32}-1]$
float (23-bit mantissa)	32	32	$[-3.402E+38, -1.175E-38]$ $[+1.175E-38, +3.402E+38]$
double long double (23-bit mantissa)	32	32	$[-3.402E+38, -1.175E-38]$ $[+1.175E-38, +3.402E+38]$
pointer to data ** pointer to function (code pointer) ** __far pointer **	32	32	$[0, 2^{14}-1]$ $[0, 2^{16}-1]$ $[0, 2^{32}-1]$

* You can use the type qualifier `__mau8` only on objects that have the `__far` qualifier, because only objects located in the FPI space can have byte access.

** Pointers are calculated using 32-bit arithmetic and compared as 14-bit values (data pointers), 16-bit values (code pointers) or 32-bit values (`__far` pointers).

Aggregate and Union Types

Aggregate types are aligned on 32 bits by default. All members of the aggregate types are aligned as required by their individual types as listed in the table above. The struct/union data types may contain bit-fields. The allowed bit-field fundamental data types are `_Bool`, `(un)signed char` and `(un)signed int`. The maximum bit-field size is equal to that of the type's size. For the bit-field types the same rules regarding to alignment and signed-ness apply as specified for the fundamental data types. In addition, the following rules apply:

- The first bit-field is stored at the least significant bits. Subsequent bit-fields fill the higher significant bits.
- A bit-field of a particular type cannot cross a boundary as is specified by its maximum width. For example, a bit-field of type `int` cannot cross a 32-bit boundary.
- Bit-fields share a storage unit with other bit-field members if and only if there is sufficient space in the storage unit.
- An unnamed bit-field creates a gap that has the size of the specified width. As a special case, an unnamed bit-field having width 0 (zero) prevents any further bit-field from residing in the storage unit corresponding to the type of the zero-width bit-field.

1.1.1. Changing the Alignment: `__align()`

By default the PCP compiler aligns objects to the minimum alignment required by the architecture. With the attribute `__align()` you can change the object alignment that is located in the FPI space. Objects

qualified with `__far` are located in the FPI space. The alignment must be a power of two. `__align()` has no effect on object located in the PRAM space of the PCP.

Example:

```
int __align( 8 ) __far src[4];
```

The compiler generates the following assembly:

```
.sdecl  '.bss.linear', data, linear, clear
.sect   '.bss.linear'
.global _PCP_src
.align  8
_PCP_src:      .type   object
               .size   _PCP_src,16
               .space  16
```

Instead of the attribute `__align()` you can also use `#pragma align`.

1.2. Accessing Memory

You can use static memory type qualifiers to allocate static objects in a particular part of the addressing space of the processor.

In addition, you can place variables at absolute addresses with the keyword `__at()`.

1.2.1. Memory Type Qualifiers

In the C language you can specify that a variable must lie in a specific part of memory. You can do this with a *memory type qualifier*. If you do not specify a memory type qualifier, data objects get a default memory type.

You can specify the following memory type qualifiers:

Qualifier	Description	Location	Maximum object size	Pointer size	Pointer arithmetic	Section type
<code>__near</code>	Data	PRAM space	64 kB	32-bit	14-bit	data
<code>__far</code> *	Far data	FPI space	4 GB	32-bit	32-bit	linear
<code>__far</code> <code>__mau8</code>	Allow 8-bit or 16-bit data allocation	FPI space	8-bit or 16-bit	32-bit	32-bit	linear

* If you do not specify `__far`, the compiler chooses where to place the declared object.

Data objects are located by default in the PRAM space of the PCP (`__near` is the default). The Memory Access Unit (MAU) of PRAM is 32-bit. All objects located in the PRAM always have a size of 32 bits.

TASKING VX-toolset for PCP User Guide

Data objects that are qualified `__far` are located in the FPI space. The FPI space is the TriCore[®] linear address space. The Memory Access Unit (MAU) of the FPI is 8-bit. By default the object size of `__far` qualified objects is 32-bit, because the default data type size is 32-bits on the PCP for PRAM and FPI.

`__far` data objects with type `char` or `short` can have type modifier `__mau8` to allow 8-bit and 16-bit data allocation on the FPI. FPI instructions are generated to access objects that are qualified `__far`.

Examples

```
char c; // 32-bit object in PRAM
short s; // 32-bit object in PRAM
int i; // 32-bit object in PRAM
char text[] = "No smoking"; // 11 words in PRAM

__far char c; // 32-bit object in FPI
__far short s; // 32-bit object in FPI
__far int i; // 32-bit object in FPI
__far char text[] = "No smoking"; // 11 words in FPI

__far __mau8 char c; // 8-bit object in FPI
__far __mau8 short s; // 16-bit object in FPI
__far __mau8 int i; // 32-bit object in FPI
__far __mau8 char text[] = "No smoking"; // 11 bytes in FPI
```

1.2.2. Pointers

The PCP compiler supports code and data pointers as shown in the following table.

Pointer	Location	Maximum object size	Pointer size	Section type
<code>__near</code> data pointer	PRAM space	64 kB	14-bit	data
<code>__far</code> data pointer	FPI space	4 GB	32-bit	linear
code pointer	CMEM space	128 kB	16-bit	code

The default section name is equal to the generated section type that is prefixed with `.pcptext.` for code in CMEM and `.pcpdata.` for data in PRAM. The default code section name is `.pcptext.code` and the default PRAM data section name is `.pcpdata.data`. The default FPI data section name uses a prefix conform the TriCore C compiler. E.g. `__far int i;` is located in a section with name `.bss.linear`. You can change section names with `#pragma section` or with the command line option `--rename-sections`.

Pointers with memory type qualifiers

Pointers for the PCP can have two types: a 'logical' type and a memory type. For example,

```
char __far * p;
```

means `p` has memory type PRAM (`p` itself is allocated in PRAM, PRAM is the default), but has logical type 'character in target memory space FPI (`__far`)'. The memory type qualifier used left to the `'*'`, specifies the target memory of the pointer, the memory type qualifier used right to the `'*'`, specifies the storage memory of the pointer.

Examples:

```
int *p;           // pointer 'p' located in PRAM pointing to int in PRAM
int __far *q;     // pointer 'q' located in PRAM pointing to int in FPI
int * __far r;    // pointer 'r' located in FPI pointing to int in PRAM
int __far * __far s; // pointer 's' located in FPI pointing to int in FPI
```

A PRAM pointer cannot be converted to an FPI (`__far`) pointer or visa versa.

1.2.3. Placing an Object at an Absolute Address: `__at()`

Just like you can declare a variable in a specific part of memory (using memory type qualifiers), you can also place an object or function at an absolute address in memory.

With the attribute `__at()` you can specify an absolute address. The address is a 32-bit linear address.

Examples

```
unsigned char Display[80*24] __at( 0x2000 );
```

The array `Display` is placed at address `0x2000`. In the generated assembly, an absolute section is created. On this position space is reserved for the variable `Display`.

```
int i __at(0x1000) = 1;
```

The variable `i` is placed at address `0x1000` and is initialized.

```
void f(void) __at( 0xf0ff + 1 ) { }
```

The function `f` is placed at address `0xf100`.

Restrictions

Take note of the following restrictions if you place a variable at an absolute address:

- The argument of the `__at()` attribute must be a constant address expression.
- You can place only global variables at absolute addresses. Parameters of functions, or automatic variables within functions cannot be placed at absolute addresses.
- A variable that is declared `extern`, is not allocated by the compiler in the current module. Hence it is not possible to use the keyword `__at()` on an external variable. Use `__at()` at the definition of the variable.
- You cannot place structure members at an absolute address.
- Absolute variables cannot overlap each other. If you declare two absolute variables at the same address, the assembler and/or linker issues an error. The compiler does not check this.

1.2.4. Accessing Hardware from C

Using Special Function Registers

It is easy to access Special Function Registers (SFRs) that relate to peripherals from C. The SFRs are defined in a special function register file (*.sfr) as symbol names for use with the compiler. An SFR file contains the names of the SFRs and the bits in the SFRs.

Example use in C (SFRs from `regtc1165.sfr`):

```
void set_sfr(void)
{
    LBCU_SRC.I |= 0xb32a; /* access LBCU Service Request
                          Control register as a whole */

    LBCU_SRC.B.SRE = 0x1; /* access SRE bit-field of LBCU
                          Service Request Control register */
}
```

You can find a list of defined SFRs and defined bits by inspecting the SFR file for a specific processor. The files are named `regcpu.sfr`, where `cpu` is the CPU specified with the C compiler option `--cpu` (you can use the compiler option `--cpu=tc1165` to compile the example above). The compiler automatically includes this register file, unless you specify option `--no-tasking-sfr`. The files are located in the standard `include` directory.

Defining Special Function Registers: `__sfrbit32`

SFRs are defined in SFR files and are written in C. With the data type qualifier `__sfrbit32` you can declare bit-fields in special function registers.

According to the *TriCore Embedded Applications Binary Interface*, 'normal' bit-fields are accessed as `char`, `short` or `int`. Bit-fields are aligned according to the table in [Section 1.1, Data Types](#).

If you declare bit-fields in special function registers, this behavior is not always desired: some special function registers require 32-bit access. To force 32-bit access, you can use the data type qualifier `__sfrbit32`.

When the SFR contains fields, the layout of the SFR is defined by a typedef-ed union. The next example is part of an SFR file and illustrates the declaration of a special register using the data type qualifier `__sfrbit32`:

```
typedef volatile union
{
    struct
    {
        unsigned __sfrbit32 SRPN : 8; /* Service Priority Number */
        unsigned __sfrbit32      : 2;
        unsigned __sfrbit32 TOS  : 2; /* Type-of-Service Control */
        unsigned __sfrbit32 SRE  : 1; /* Service Request Enable Control */
        unsigned __sfrbit32 SRR  : 1; /* Service Request Flag */
        unsigned __sfrbit32 CLRR : 1; /* Request Flag Clear Bit */
    };
};
```

```

unsigned __sfrbit32 SETR : 1; /* Request Flag Set Bit */
unsigned __sfrbit32      : 16;
} B;

int I;
unsigned int U;
} LBCU_SRC_type;

```

Read-only fields can be marked by using the `const` keyword.

The SFR is defined by a cast to a 'typedef-ed union' pointer. The SFR address is given in parenthesis. Read-only SFRs are marked by using the `const` keyword in the macro definition.

```

#define LBCU_SRC (*(LBCU_SRC_type*)(0xF87FFFCu))
                /* LBCU Service Control Register */

```

Restrictions

- You can use the `__sfrbit32` data type qualifier only for `int` types. The compiler issues an error if you use for example `__sfrbit32 char x : 8;`
- When you use the `__sfrbit32` data type qualifier for other types than a bit-field, the compiler ignores this without a warning. For example, `__sfrbit32 int global;` is equal to `int global;`.
- Structures or unions that contain a member qualified with `__sfrbit32`, are zero padded to complete a full word if necessary. The structure or union will be word aligned.

1.3. Using Assembly in the C Source: `__asm()`

With the keyword `__asm` you can use assembly instructions in the C source and pass C variables as operands to the assembly code. Be aware that C modules that contain assembly are not portable and harder to compile in other environments.

The compiler does not interpret assembly blocks but passes the assembly code to the assembly source file; they are regarded as a black box. So, it is your responsibility to make sure that the assembly block is syntactically correct. Possible errors can only be detected by the assembler.

General syntax of the `__asm` keyword

```

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

```

<i>instruction_template</i>	Assembly instructions that may contain parameters from the input list or output list in the form: <code>%param_nr</code>
<code>%param_nr</code>	Parameter number in the range 0 .. 9.
<i>output_param_list</i>	<code>[["[&]constraint_char"(C_expression)],...</code>
<i>input_param_list</i>	<code>[["constraint_char"(C_expression)],...</code>

&	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.
<i>constraint_char</i>	Constraint character: the type of register to be used for the <i>C_expression</i> . See the table below.
<i>C_expression</i>	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.
<i>register_save_list</i>	[[" <i>register_name</i> "],...]
<i>register_name</i>	Name of the register you want to reserve. Note that saving too much registers can make register allocation impossible.

Specifying registers for C variables

With a *constraint character* you specify the register type for a parameter.

You can reserve the registers that are used in the assembly instructions, either in the parameter lists or in the reserved register list (*register_save_list*). The compiler takes account of these lists, so no unnecessary register saves and restores are placed around the inline assembly instructions.

Constraint character	Type	Operand	Remark
r	register	r0 .. r7	
<i>number</i>	type of operand it is associated with	same as <i>%number</i>	Input constraint only. The <i>number</i> must refer to an output parameter. Indicates that <i>%number</i> and <i>number</i> are the same register.

Loops and conditional jumps

The compiler does not detect loops with multiple `__asm()` statements or (conditional) jumps across `__asm()` statements and will generate incorrect code for the registers involved.

If you want to create a loop with `__asm()`, the whole loop must be contained in a single `__asm()` statement. The same counts for (conditional) jumps. As a rule of thumb, all references to a label in an `__asm()` statement must be in that same statement. You can use numeric labels for these purposes.

Example 1: no input or output

A simple example without input or output parameters. You can use any instruction or label. When it is required that a sequence of `__asm()` statements generates a contiguous sequence of instructions, then they can be best combined to a single `__asm()` statement. Compiler optimizations can insert instruction(s) in between `__asm()` statements. Use newline characters '\n' to continue on a new line in a `__asm()` statement.

```
__asm( "nop\n"
      "nop" );
```


Example 2: using output parameters

Assign the result of inline assembly to a variable. With the constraint `r` a register is chosen for the parameter; the compiler decides which register it uses. The `%0` in the instruction template is replaced with the name of this register. The compiler generates code to assign the result to the output variable.

```
int out;
void addone( void )
{
    __asm( "add.i %0,#1"
          : "=r" (out) );
}
```

Generated assembly code:

```
add.i r5,#1
ldl.il r7,@DPTR(_PCP_out)
st.pi r5,[_PCP_out]
```

Example 3: using input parameters

Assign a variable to a register. A register is chosen for the parameter because of the constraint `r`; the compiler decides which register is best to use. The `%0` in the instruction template is replaced with the name of this register. The compiler generates code to move the input variable to the input register. Because there are no output parameters, the output parameter list is empty. Only the colon has to be present.

```
int in;
void initreg( void )
{
    __asm( "MOV R0,%0,cc_Z"
          :
          : "r" (in) );
}
```

Generated assembly code:

```
ldl.il r7,@DPTR(_PCP_in)
ld.pi r5,[_PCP_in]
MOV R0,r5,cc_Z
```

Example 4: using input and output parameters

Multiply two C variables and assign the result to a third C variable. Registers are necessary for the input and output parameters (constraint `r`, `%0` for `out`, `%1` for `in1`, `%2` for `in2` in the instruction template). The compiler generates code to move the input expressions into the input registers and to assign the result to the output variable.

```
int in1, in2, out;

void multiply( void )
{
```

```
__asm( "minit\t%1,%2\n"  
      "\tmstep.u\t%0,%2\n"  
      "\tmstep.u\t%0,%2\n"  
      "\tmstep.u\t%0,%2\n"  
      "\tmstep.u\t%0,%2"  
      : "=r" (out)  
      : "r" (in1), "r" (in2) );  
}
```

Generated assembly code:

```
_PCP_multiply: .type func  
ldl.il r7,@DPTR(_PCP_in1)  
ld.pi r5,[_PCP_in1]  
ld.pi r1,[_PCP_in2]  
minit r5,r1  
mstep.u r5,r1  
mstep.u r5,r1  
mstep.u r5,r1  
mstep.u r5,r1  
  
ldl.il r7,@DPTR(_PCP_out)  
st.pi r5,[_PCP_out]
```

Example 5: reserving registers

Sometimes an instruction knocks out certain specific registers. The most common example of this is a function call, where the called function is allowed to do whatever it likes with some registers. If this is the case, you can list specific registers that get clobbered by an operation after the inputs.

Same as *Example 4*, but now register `r1` is a reserved register. You can do this by adding a reserved register list (`: "r1"`). As you can see in the generated assembly code, register `r1` is not used (register `r3` is used instead).

```
int in1, in2, out;  
  
void multiply( void )  
{  
__asm( "minit\t%1,%2\n"  
      "\tmstep.u\t%0,%2\n"  
      "\tmstep.u\t%0,%2\n"  
      "\tmstep.u\t%0,%2\n"  
      "\tmstep.u\t%0,%2"  
      : "=r" (out)  
      : "r" (in1), "r" (in2)  
      : "r1" );  
}
```

Generated assembly code:

```

_PCP_multiply: .type   func
               ldl.il  r7,@DPTR(_PCP_in1)
               ld.pi   r5,[_PCP_in1]
               ld.pi   r3,[_PCP_in2]
               minit   r5,r3
               mstep.u r5,r3
               mstep.u r5,r3
               mstep.u r5,r3
               mstep.u r5,r3

               ldl.il  r7,@DPTR(_PCP_out)
               st.pi   r5,[_PCP_out]

```

Example 6: input and output are the same

If the input and output must be the same you must use a number constraint. The following example inverts the value of the input variable `invar` and returns this value to `outvar`. Parameter `%0` corresponds to `outvar`. To indicate that `invar` uses the same register as `outvar`, the input constraint `'0'` is used which indicates that `invar` also corresponds to `%0`.

```

int outvar;

static inline void invert(int invar)
/* 'static inline' makes assembly easier to read */
{
    __asm ("not %0,%1,cc_SGT": "=r"(outvar): "0"(invar) );
}

void main(void)
{
    invert(255);
}

```

Generated assembly code:

```

ld.i    r5,0x3f
ldl.il  r5,0xff
not     r5,r5,cc_SGT
ldl.il  r7,@DPTR(_PCP_outvar)
st.pi   r5,[_PCP_outvar]

```

1.4. Attributes

You can use the keyword `__attribute__` to specify special attributes on declarations of variables, functions, types, and fields. The `alias`, `always_inline`, `const`, `export`, `format`, `malloc`, `noinline`, `noreturn`, `pure`, `section`, `unused`, `used`, and `weak` attributes are supported.

Syntax:

```
__attribute__((name,...))
```

TASKING VX-toolset for PCP User Guide

or:

```
__name__
```

The second syntax allows you to use attributes in header files without being concerned about a possible macro of the same name.

alias("symbol")

You can use `__attribute__((alias("symbol")))` to specify that the function declaration appears in the object file as an alias for another symbol. For example:

```
void __f() { /* function body */; }  
void f() __attribute__((weak, alias("__f")));
```

declares 'f' to be a weak alias for '__f'.

const

You can use `__attribute__((const))` to specify that a function has no side effects and will not access global data. This can help the compiler to optimize code.

The following kinds of functions should not be declared `__const__`:

- A function with pointer arguments which examines the data pointed to.
- A function that calls a non-const function.

export

You can use `__attribute__((export))` to specify that a variable/function has external linkage and should not be removed. During MIL linking, the compiler treats external definitions at file scope as if they were declared `static`. As a result, unused variables/functions will be eliminated, and the alias checking algorithm assumes that objects with static storage cannot be referenced from functions outside the current module. During MIL linking not all uses of a variable/function can be known to the compiler. For example when a variable is referenced in an assembly file or a (third-party) library. With the `export` attribute the compiler will not perform optimizations that affect the unknown code.

```
int i __attribute__((export)); /* 'i' has external linkage */
```

format(type,arg_string_index,arg_check_start)

You can use `__attribute__((format(type,arg_string_index,arg_check_start)))` to specify that functions take format strings as arguments and that calls to these functions must be type-checked against a format string, similar to the way the compiler checks calls to the functions `printf`, `scanf`, `strftime`, and `strfmon` for errors.

`arg_string_index` is a constant integral expression that specifies which argument in the declaration of the user function is the format string argument.

`arg_check_start` is a constant integral expression that specifies the first argument to check against the format string. If there are no arguments to check against the format string (that is, diagnostics should only

be performed on the format string syntax and semantics), `arg_check_start` should have a value of 0. For `strftime`-style formats, `arg_check_start` must be 0.

Example:

```
int foo(int i, const char *my_format, ...) __attribute__((format(printf, 2, 3)));
```

The format string is the second argument of the function `foo` and the arguments to check start with the third argument.

malloc

You can use `__attribute__((malloc))` to improve optimization and error checking by telling the compiler that:

- The return value of a call to such a function points to a memory location or can be a null pointer.
- On return of such a call (before the return value is assigned to another variable in the caller), the memory location mentioned above can be referenced only through the function return value; e.g., if the pointer value is saved into another global variable in the call, the function is not qualified for the `malloc` attribute.
- The lifetime of the memory location returned by such a function is defined as the period of program execution between a) the point at which the call returns and b) the point at which the memory pointer is passed to the corresponding deallocation function. Within the lifetime of the memory object, no other calls to `malloc` routines should return the address of the same object or any address pointing into that object.

noinline

You can use `__attribute__((noinline))` to prevent a function from being considered for inlining. Same as keyword `__noinline` or `#pragma noinline`.

always_inline

With `__attribute__((always_inline))` you force the compiler to inline the specified function, regardless of the optimization strategy of the compiler itself. Same as keyword `inline` or `#pragma inline`.

noreturn

Some standard C function, such as `abort` and `exit` cannot return. The C compiler knows this automatically. You can use `__attribute__((noreturn))` to tell the compiler that a function never returns. For example:

```
void fatal() __attribute__((noreturn));

void fatal( /* ... */ )
{
    /* Print error message */
    exit(1);
}
```

The function `fatal` cannot return. The compiler can optimize without regard to what would happen if `fatal` ever did return. This can produce slightly better code and it helps to avoid warnings of uninitialized variables.

protect

You can use `__attribute__((protect))` to exclude a variable/function from the duplicate/unreferenced section removal optimization in the linker. When you use this attribute, the compiler will add the "protect" section attribute to the symbol's section. Example:

```
int i __attribute__((protect));
```

Note that the `protect` attribute will not prevent the compiler from removing an unused variable/function (see the `used` symbol attribute).

This attribute is the same as `#pragma protect/endprotect`.

pure

You can use `__attribute__((pure))` to specify that a function has no side effects, although it may read global data. Such pure functions can be subject to common subexpression elimination and loop optimization.

section("section_name")

You can use `__attribute__((section("name")))` to specify that a function must appear in the object file in a particular section. For example:

```
extern void foobar(void) __attribute__((section("bar")));
```

puts the function `foobar` in the section named `bar`.

See also `#pragma section`.

used

You can use `__attribute__((used))` to prevent an unused symbol from being removed, by both the compiler and the linker. Example:

```
static const char copyright[] __attribute__((used)) = "Copyright 2010 Altium BV";
```

When there is no C code referring to the `copyright` variable, the compiler will normally remove it. The `__attribute__((used))` symbol attribute prevents this. Because the linker should also not remove this symbol, `__attribute__((used))` implies `__attribute__((protect))`.

unused

You can use `__attribute__((unused))` to specify that a variable or function is possibly unused. The compiler will not issue warning messages about unused variables or functions.

weak

You can use `__attribute__((weak))` to specify that the symbol resulting from the function declaration or variable must appear in the object file as a weak symbol, rather than a global one. This is primarily useful when you are writing library functions which can be overwritten in user code without causing duplicate name errors.

See also `#pragma weak`.

1.5. Pragmas to Control the Compiler

Pragmas are keywords in the C source that control the behavior of the compiler. Pragmas overrule compiler options. Put pragmas in your C source where you want them to take effect. Unless stated otherwise, a pragma is in effect from the point where it is included to the end of the compilation unit or until another pragma changes its status.

The syntax is:

```
#pragma pragma-spec pragma-arguments [on | off | default | restore]
```

or:

```
_Pragma( "pragma-spec pragma-arguments [on | off | default | restore]" )
```

Some pragmas can accept the following special arguments:

<code>on</code>	switch the flag on (same as without argument)
<code>off</code>	switch the flag off
<code>default</code>	set the pragma to the initial value
<code>restore</code>	restore the previous value of the pragma

Some pragmas have an equivalent command line option. This is useful if you want to overrule certain keywords in the C source without the need to change the C source itself.

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

alias *symbol=defined_symbol*

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

align {*value* | default | restore}

Change the alignment of objects located in the FPI space. By default the PCP compiler aligns objects to the minimum alignment required by the architecture. With this pragma you can increase this alignment for objects of four bytes or larger. The value must be a power of two.

See [Section 1.1.1, Changing the Alignment: `__align\(\)`](#).

clear / noclear

By default, uninitialized global or static variables are cleared to zero on startup. With `pragma noclear`, this step is skipped. Pragma `clear` resumes normal behavior. This pragma applies to constant data as well as non-constant data.

See [C compiler option --no-clear](#).

compactmaxmatch {value | default | restore}

With this pragma you can control the maximum size of a match.

See [C compiler option --compact-max-size](#).

extension isuffix [on | off | default | restore]

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

```
float 0.5i
```

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 can force an external reference (`.EXTERN` assembler directive), even when the `symbol` is not used in the module.

inline / noinline / smartinline

See [Section 1.8.3, Inlining Functions: inline](#).

inline_max_incr {value | default | restore}

inline_max_size {value | default | restore}

With these pragmas you can control the automatic function inlining optimization process of the compiler. It has effect only when you have enable the inlining optimization ([C compiler option --optimize=+inline](#)).

See [C compiler options --inline-max-incr / --inline-max-size](#).

linear_switch / jump_switch / binary_switch / 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.
```


<code>jump_switch</code>	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.
<code>binary_switch</code>	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.
<code>smart_switch</code>	let the compiler decide the switch method used

See also [Section 1.7, *Switch Statement*](#).

macro / nomacro [on | off | default | restore]

Turns macro expansion on or off. By default, macro expansion is enabled.

message "message" ...

Print the message string(s) on standard output.

nomisrac [*nr*,...] [default | restore]

Without arguments, this pragma disables MISRA-C checking. Alternatively, you can specify a comma-separated list of MISRA-C rules to disable.

See [C compiler option `--misrac`](#) and [Section 3.7.2, *C Code Checking: MISRA-C*](#).

novector *value* [default | restore]

With this pragma you tell the compiler not to generate code for channel vectors and channel context.

See [C compiler option `--no-vector`](#).

optimize [*flags* | default | restore] / endoptimize

You can overrule the C compiler option `--optimize` for the code between the pragmas `optimize` and `endoptimize`. The pragma works the same as [C compiler option `--optimize`](#).

See [Section 3.6, *Compiler Optimizations*](#).

protect [on | off | default | restore] / endprotect

With these pragmas you can protect sections against linker optimizations. This excludes a section from unreferenced section removal and duplicate section removal by the linker. `endprotect` restores the default section protection.

section [*type=name* | default | restore] / endsection

Changes section names. See [Section 1.10, *Compiler Generated Sections*](#) and [C compiler option `--rename-sections`](#) for more information.

source [on | off | default | restore] / nosource

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

See [C compiler option --source](#).

stdinc [on | off | default | restore]

This pragma changes the behavior of the `#include` directive. When set, the C compiler options `--include-directory` and `--no-stdinc` are ignored.

tradeoff {/level/ | default | restore}

Specify tradeoff between speed (0) and size (4). See [C compiler option --tradeoff](#)

warning [number,...] [default | restore]

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

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.

1.6. Predefined Preprocessor Macros

The TASKING C compiler supports the predefined macros as defined in the table below. The macros are useful to create conditional C code.

Macro	Description
<code>__BIG_ENDIAN__</code>	Expands to 0. The processor accesses data in little-endian.
<code>__BUILD__</code>	Identifies the build number of the compiler, composed of decimal digits for the build number, three digits for the major branch number and three digits for the minor branch number. For example, if you use build 1.22.1 of the compiler, <code>__BUILD__</code> expands to 1022001. If there is no branch number, the branch digits expand to zero. For example, build 127 results in 127000000.
<code>__CORE__</code>	Expands to the name of the core depending on the C compiler options <code>--cpu</code> and <code>--core</code> . The symbol expands to <code>pcp2</code> when no <code>--cpu</code> and no <code>--core</code> is supplied.

Macro	Description
<code>__CORE_core__</code>	A symbol is defined depending on the options <code>--cpu</code> and <code>--core</code> . The <i>core</i> is converted to upper case. For example, if <code>--cpu=tc1165</code> is specified, the symbol <code>__CORE_PCP2__</code> is defined. When no <code>--core</code> or <code>--cpu</code> is supplied, the compiler defines <code>__CORE_PCP2__</code> .
<code>__CPU__</code>	Expands to the name of the CPU supplied with the option <code>--cpu</code> . When no <code>--cpu</code> is supplied, this symbol is not defined. For example, if <code>--cpu=tc1165</code> is specified, the symbol <code>__CPU__</code> expands to <code>tc1165</code> .
<code>__CPU_cpu__</code>	A symbol is defined depending on the option <code>--cpu=cpu</code> . The <i>cpu</i> is converted to uppercase. For example, if <code>--cpu=tc1165</code> is specified, the symbol <code>__CPU_TC1165__</code> is defined. When no <code>--cpu</code> is supplied, this symbol is not defined.
<code>__DATE__</code>	Expands to the compilation date: "mmm dd yyyy".
<code>__FILE__</code>	Expands to the current source file name.
<code>__LINE__</code>	Expands to the line number of the line where this macro is called.
<code>__REVISION__</code>	Expands to the revision number of the compiler. Digits are represented as they are; characters (for prototypes, alphas, betas) are represented by -1. Examples: <code>v1.0r1 -> 1</code> , <code>v1.0rb -> -1</code>
<code>__SFRFILE__(cpu)</code>	This macro expands to the filename of the used SFR file, including the <code><></code> . The <i>cpu</i> is the argument of the macro. For example, if <code>--cpu=tc1165</code> is specified, the macro <code>__SFRFILE__(__CPU__)</code> expands to <code>__SFRFILE__(tc1165)</code> , which expands to <code><regtc1165.sfr></code> .
<code>__SINGLE_FP__</code>	Expands to 1 ('double' is always treated as 'float').
<code>__STDC__</code>	Identifies the level of ANSI standard. The macro expands to 1 if you set option <code>--language</code> (Control language extensions), otherwise expands to 0.
<code>__STDC_HOSTED__</code>	Always expands to 0, indicating the implementation is not a hosted implementation.
<code>__STDC_VERSION__</code>	Identifies the ISO-C version number. Expands to 199901L for ISO C99 or 199409L for ISO C90.
<code>__TASKING__</code>	Identifies the compiler as a TASKING compiler. Expands to 1 if a TASKING compiler is used.
<code>__TASKING_SFR__</code>	Expands to 1 if TASKING <code>.sfr</code> files are used. Not defined when option <code>--no-tasking-sfr</code> is used.
<code>__TIME__</code>	Expands to the compilation time: "hh:mm:ss"
<code>__VERSION__</code>	Identifies the version number of the compiler. For example, if you use version 2.1r1 of the compiler, <code>__VERSION__</code> expands to 2001 (dot and revision number are omitted, minor version number in 3 digits).

Example

```
#ifdef __CORE_PCP1__
/* this part is only valid for a PCP1 core */
```

```
...  
#endif
```

1.7. Switch Statement

The TASKING C compiler supports three ways of code generation for a switch statement: a jump chain (linear switch), a jump table or a binary search table.

A *jump chain* is comparable with an if/else-if/else-if/else construction. 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. A *binary search table* is a table filled with a value to compare the switch argument with and a target address to jump to.

By default, the compiler will automatically choose the most efficient switch implementation based on code and data size and execution speed. With the C compiler option `--tradeoff` you can tell the compiler to emphasis more on speed than on ROM size.

Especially for large switch statements, the jump table approach executes faster than the lookup table approach. Also the jump table has a predictable behavior in execution speed: independent of the switch argument, every case is reached in the same execution time. However, when the case labels are distributed far apart, the jump table becomes sparse, wasting code memory. The compiler will not use the jump table method when the waste becomes excessive.

With a small number of cases, the jump chain method can be faster in execution and shorter in size.

How to overrule the default switch method

You can overrule the compiler chosen switch method by using a pragma:

```
#pragma linear_switch   force jump chain code  
#pragma jump_switch     force jump table code  
#pragma binary_switch   force binary search table code  
#pragma smart_switch    let the compiler decide the switch method used (this is the default)
```

The switch pragmas must be placed before the `switch` statement. Nested `switch` statements use the same switch method, unless the nested `switch` is implemented in a separate function which is preceded by a different switch pragma.

Example:

```
/* place pragma before function body */  
  
#pragma jump_switch  
  
void test(unsigned char val)  
{ /* function containing the switch */  
  switch (val)  
  {  
    /* use jump table */  
  }  
}
```

```

    }
}

```

1.8. Functions

1.8.1. Calling Convention

Parameter Passing

A lot of execution time of an application is spent transferring parameters between functions. The fastest parameter transport is via registers. Therefore, function parameters are first passed via registers. If no more registers are available for a parameter, the compiler pushes parameters on the stack.

Registers available for parameter passing are R1, R3, R4, R6, R0. The parameters are processed from left to right. The first unused register is used. Registers are searched for in the order listed above. When a parameter is larger than 32 bit, or when all registers are used, parameter passing continues on the stack. The stack grows from higher towards lower addresses, each parameter on the stack is stored in little endian. The alignment on the stack depends on the data type as listed in [Section 1.1, Data Types](#).

Structures up to four bytes are passed via a register. Larger structures are passed via the stack.

The PCP compiler uses a static stack, which restricts the number of arguments passed for an indirect function call. Parameters of an indirect function call can only be passed in registers and not via the static stack.

Example with three arguments:

```
func1( int a, long b, char c )
```

a (first parameter) is passed in register R1.

b (second parameter) is passed in register R3.

c (third parameter) is passed in register R4.

Variable Argument Lists

For functions with a variable argument list, the last fixed parameter and all subsequent parameters must be pushed on the stack. For parameters before the last fixed parameter the normal parameter passing rules apply.

Variable arguments are not supported for indirect function calls, due to the static stack implementation.

Function Return Values

The C compiler uses register R1 to store C function return values.

When the function return type is a structure, it is copied to a "return area" that is allocated by the caller. The address of this area is passed as an implicit first argument in R6.

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 overlaid by the linker using a call graph.

1.8.2. Register Usage

The PCP C compiler uses registers according to the convention given in the following table.

Register	Class	Purpose
R0	caller saves	Parameter passing and automatic variables
R1	caller saves	Parameter passing, automatic variables and return values
R2	callee saves	Automatic variables, stack frame pointer and function return address
R3	caller saves	Parameter passing and automatic variables
R4	caller saves	Parameter passing and automatic variables
R5	caller saves	Automatic variables and function return address of code compaction functions
R6	caller saves	Parameter passing, automatic variables and return buffer
R7	special purpose	PC, CC, DPTR

The registers are classified: *caller saves*, *callee saves* and *special purpose*.

caller saves These registers are allowed to be changed by a function without saving the contents. Therefore, the calling function must save these registers when necessary prior to a function call.

callee saves These registers must be saved by the called function, i.e. the caller expects them not to be changed after the function call.

special purpose The purpose of R7 is defined by the PCP core.

1.8.3. Inlining Functions: inline

With the C compiler option `--optimize=+inline`, the C compiler automatically inlines small functions in order to reduce execution time (smart inlining). The compiler inserts the function body at the place the function is called. The C compiler decides which functions will be inlined. You can overrule this behavior with the two keywords `inline` (ISO-C) and `__noinline`.

With the `inline` keyword you force the compiler to inline the specified function, regardless of the optimization strategy of the compiler itself:

```
inline unsigned int abs(int val)
{
    unsigned int abs_val = val;
    if (val < 0) abs_val = -val;
}
```

```

    return abs_val;
}

```

If a function with the keyword `inline` is not called at all, the compiler does not generate code for it.

You must define inline functions in the same source module as in which you call the function, because the compiler only inlines a function in the module that contains the function definition. When you need to call the inline function from several source modules, you must include the definition of the inline function in each module (for example using a header file).

With the `__noinline` keyword, you prevent a function from being inlined:

```

__noinline unsigned int abs(int val)
{
    unsigned int abs_val = val;
    if (val < 0) abs_val = -val;
    return abs_val;
}

```

Using pragmas: inline, noinline, smartinline

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

```

#pragma inline
unsigned int abs(int val)
{
    unsigned int abs_val = val;
    if (val < 0) abs_val = -val;
    return abs_val;
}
#pragma noinline
void main( void )
{
    int i;
    i = abs(-1);
}

```

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

With the `#pragma noinline`/`#pragma smartinline` you can temporarily disable the default behavior that the C compiler automatically inlines small functions when you turn on the [C compiler option `--optimize+=inline`](#).

With the [C compiler options `--inline-max-incr`](#) and `--inline-max-size` you have more control over the automatic function inlining process of the compiler.

Combining inline with `__asm` to create intrinsic functions

With the keyword `__asm` it is possible to use assembly instructions in the body of an inline function. Because the compiler inserts the (assembly) body at the place the function is called, you can create your own intrinsic function. See [Section 1.8.5, Intrinsic Functions](#).

1.8.4. Interrupt Functions

The PCP has an unusual programming model. The best way to think of PCP programming is that there is a series of autonomous programs, or tasks that are called *channel programs*. These can be very short and simple, or very complex and long, and these can be mixed together. The PCP has a channel program associated with each interrupt number (SRPN). This could be thought of as the interrupt routine for a given interrupt source. When an interrupt of number “n” is received by the PCP core, it restores the context associated with number “n” from PRAM, and begins executing Channel Program “n” from Code Memory until it encounters a terminating condition - usually an EXIT instruction. At that point it saves the current context for number “n” back into the PRAM. If there is a new pending interrupt it starts this process again. If there is no pending new interrupt, the PCP stops until there is a new interrupt to process.

The PCP C compiler only supports the Full Context Model. Full context means that there are eight registers per channel available for the compiler.

For an extensive description of the PCP channel operation, see chapter *Peripheral Control Processor (PCP)* in the *User's Manual* of the TriCore [Infineon].

1.8.4.1. Defining an Interrupt Service Routine: `__interrupt()`

With the function type qualifier `__interrupt()` you can declare a function as an interrupt function (interrupt service routine or channel program). The function type qualifier `__interrupt()` takes one channel number (0..255) as argument.

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

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

The argument *channel_number* is an 8-bit channel number that defines the channel entry table address and the context address. The channel number must be in range [0..255]. Channel number 0 is not used on the PCP; for interrupts with channel number 0 the channel entry table and the channel context are not generated.

For "Channel Start at Context PC" mode (CS.RCB=0) the compiler generates a section containing the context of the appropriate channel. The PC context (R7.PC) is initialized with the start address of interrupt service routine. The Channel Enable (R7.CEN) context is set to 1. The Enable Interrupt Control context is by default set to zero, because a channel cannot be interrupted by another channel. With the **C compiler option `--interrupt-enable`** you can set the Interrupt Control context to allow the channel to be interrupted. Channels that have interrupts enabled must be linked separately, because they cannot share static stack. See [Section 1.9, PCP Code Generation](#). The remainder of the R7 context is cleared also (Z,N,C,V,CN1Z,DPTR).

All other context registers (R0..R6) are initialized to zero.

For "Channel Start at Base" mode (CS.RCB=1) the compiler generates a section with the channel entry table entry of the appropriate channel. The channel table entry contains a jump to the interrupt service routine.

At interrupt function return, an EXIT instruction is generated. The arguments of the EXIT instruction generated are: EC=0, ST=0, INT=0, EP=0, cc_UC.

Example

```
void __interrupt( 1 ) isr( void )
{
    ...
}
```

1.8.4.2. Setting the Current PCP Priority Number: `__cppn()`

With the function qualifier `__cppn()` you can define the interrupt priority of an interruptible function. `__cppn()` can only be used on functions that are qualified `__interrupt`, and the functions must be interruptible (option `--interrupt-enable`). The CPPN is superfluous for functions that have interrupts disabled (R7.IEN=0).

```
void __interrupt( channel_number ) __cppn(CPPN) isr( void )
{
    ...
}
```

The function qualifier takes one argument *CPPN*. The CPPN (Current PCP interrupt Priority Number) is an 8-bit channel interrupt priority number in the range [0..255]. The channel interrupt priority number is defined in the register context R6.CPPN. At interrupt EXIT R6.CPPN is restored to this value.

1.8.4.3. Shared Data: `__share`

Shared data between PCP channels

Global data can be shared between separately linked channels with the keyword `__share`. Only global and external variables can be qualified with the keyword `__share`.

For example, in one channel the following variable is defined:

```
int __share channel1_shared_PCP_PRAM = 0;
```

In another channel you can reference this variable as:

```
extern int __share channel1_shared_PCP_PRAM;
```

`__share` variables get application scope instead of channel scope. `__share` global variables get the `_lc_` linker prefix instead of the default `_PCP_` symbol prefix.

To link channels separately use linker option `--link-only`.

Shared data between TriCore and PCP

The `__share` qualifier is not only used for sharing data between PCP channels but also for sharing global data with the TriCore CPU.

For example, in the TriCore source `tc_main.c` of the `pcp-multi-start` example delivered with the product, the following variable is defined:

```
volatile int __far __share shared_CPU_FPI;
```

In the PCP source `channel1.c` of the `pcp-multi-ch1` example delivered with the product, this variable is referenced as:

```
extern int __far __share shared_CPU_FPI;
```

To access PCP PRAM data from the TriCore CPU you can use the keyword `__pram` in the TriCore source. Also see the linker share label `_lc_s_`.

1.8.5. Intrinsic Functions

Some specific assembly instructions have no equivalence in C. *Intrinsic functions* give the possibility to use these instructions. Intrinsic functions are predefined functions that are recognized by the compiler. The compiler generates the most efficient assembly code for these functions.

The compiler always inlines the corresponding assembly instructions in the assembly source (rather than calling it as a function). This avoids parameter passing and register saving instructions which are normally necessary during function calls.

Intrinsic functions produce very efficient assembly code. Though it is possible to inline assembly code by hand, intrinsic functions use registers even more efficiently. At the same time your C source remains very readable.

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

The following example illustrates the use of an intrinsic function and its resulting assembly code.

```
__nop();
```

The resulting assembly code is inlined rather than being called:

```
nop
```

Writing your own intrinsic function

Because you can use any assembly instruction with the `__asm()` keyword, you can use the `__asm()` keyword to create your own intrinsic functions. The essence of an intrinsic function is that it is inlined.

1. First write a function with assembly in the body using the keyword `__asm()`. See [Section 1.3, Using Assembly in the C Source: `__asm\(\)`](#)
2. Next make sure that the function is inlined rather than being called. You can do this with the function qualifier `inline`. This qualifier is discussed in more detail in [Section 1.8.3, Inlining Functions: `inline`](#).

```

int in1, in2, out;

inline void __my_mul( void )
{
    __asm( "minit\t%1,%2\n"
           "\tmstep.u\t%0,%2\n"
           "\tmstep.u\t%0,%2\n"
           "\tmstep.u\t%0,%2\n"
           "\tmstep.u\t%0,%2"
           : "=r" (out)
           : "r" (in1), "r" (in2) );
}

void main(void)
{
    // call to function __my_mul
    __my_mul();
}

```

Generated assembly code:

```

_PCP_main: .type func
; __my_mul code is inlined here
ldl.il  r7,@DPTR(_PCP_in1)
ld.pi  r5,[_PCP_in1]
ld.pi  r1,[_PCP_in2]
minit  r5,r1
mstep.u r5,r1
mstep.u r5,r1
mstep.u r5,r1
mstep.u r5,r1

ldl.il  r7,@DPTR(_PCP_out)
st.pi  r5,[_PCP_out]

```

As you can see, the generated assembly code for the function `__my_mul` is inlined rather than called.

Supported intrinsic functions

You can use the following intrinsic functions in your C source:

`__alloc`

```
__alloc_t volatile __alloc( __size_t size );
```

Allocate memory. Same as C library function `malloc()`. Returns a pointer to space in external memory of `size` bytes length. Returns NULL if there is not enough space left. This function is used internally for variable length arrays, it is not to be used by end users.

`__bcopy`

```
void volatile __bcopy( void __far * dst, void __far * src,
    const signed int DSTINCDEC, const signed int SRCINCDEC,
    const unsigned int CNC, const unsigned int CNT0,
    unsigned int CNT1 );
```

This intrinsic generates a BCOPY instruction. The BCOPY instruction moves a block of data (always 32-bit data) from source (`src=R[4]`) location on the FPI bus to destination (`dst= R[5]`) on the FPI bus. Constant arguments `DSTINCDEC` and `SRCINCDEC` determine if source and destination pointer are incremented or decremented, 1 for increment, -1 for decrement and 0 for no change. Constant arguments `CNC` and `CNT0` conform to the argument values of the BCOPY instruction. `CNT1` is loaded in R6. `CNT1` if the value of `CNC` is 1 or 2. If `CNC` is 1 this intrinsic generates an outer loop.

The BCOPY instruction uses FPI Burst mode (`CNT0=2, 4 or 8 words`), which requires alignment of source and destination data blocks. Use the attribute `__align({8|16|32})` to specify the data block alignment. E.g. `unsigned int __align(8) __far array[2];`

Example:

```
int __align(8) __far src[4], dst[4];
void bcopy( void )
{
    // bcopy dst, src, +, +, CNC=2, CNT0=2, CNT1=2
    __bcopy( dst, src, 1, 1, 2, 2, 2 );
}
```

generates:

```
ldl.iu r5,@HI(_PCP_dst)
ldl.il r5,@LO(_PCP_dst)
ldl.iu r4,@HI(_PCP_src)
ldl.il r4,@LO(_PCP_src)
ld.i r6,0x2
bcopy dst+,src+,cnc=0x2,cnt0=0x2
```

`__cen`

```
void __cen( const unsigned int value );
```

Set or clear the CEN flag. You can use this intrinsic safely only when the R7 flags are preserved by the compiler (`--preserve-r7-flags`).

`__copy`

```
void volatile __copy( void __far * dst, void __far * src,
    const signed int DSTINCDEC, const signed int SRCINCDEC,
    const unsigned int CNC, const unsigned int CNT0,
    unsigned int CNT1, const unsigned int SIZE );
```

This intrinsic generates a COPY instruction. The COPY instruction moves content of source (`src=R[4]`) location on the FPI bus to destination (`dst= R[5]`) on the FPI bus. Constant arguments `DSTINCDEC`

and SRCINCDEC determine if source and destination pointer are incremented or decremented, 1 for increment, -1 for decrement and 0 for no change. Constant arguments CNC, CNT0 and SIZE are conform the argument values of the COPY instruction. CNT1 is loaded in R6.CNT1 if the value of CNC is 1 or 2. If CNC is 1 this intrinsic generates a outer loop.

Example:

```
int __far src[4], dst[4];
void copy( void )
{
    // copy dst, src, +, +, CNC=1, CNT0=2, CNT1=8 SIZE=8
    __copy( dst, src, 1, 1, 1, 2, 8, 8 );
}
```

generates:

```
    ldl.iu  r5,@HI(_PCP_dst)
    ldl.il  r5,@LO(_PCP_dst)
    ldl.iu  r4,@HI(_PCP_src)
    ldl.il  r4,@LO(_PCP_src)
    ld.i    r6,0x8
_loop:
    copy    dst+,src+,cnc=0x1,cnt0=0x2,size=0x8
    jg      _loop,cc_cnn
```

__debug

```
void __debug( const unsigned int eda, const unsigned int dac,
              const unsigned int rta, const unsigned int sdb );
```

This intrinsic generates a DEBUG instruction for the PCP2. The generated DEBUG instruction unconditionally cause a debug event. Optionally stop the channel execution (sdb=1), generate an external debug event (eda=1), disable further channel invocation (rta=0) or disable the PCP for operation (dac=1).

Example:

```
void __interrupt( 4 ) channel_4( void )
{
    /* DEBUG instruction that stops unconditionally the PCP,
     * prevents the serving of any other interrupt and generates
     * an "Illegal Operation Error".
     */
    __debug(1,1,0,1);
}
```

generates:

```
    debug eda=0x1,sdb=0x1,dac=0x1,rta=0x0,cc_uc
```

__dotdotdot

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

TASKING VX-toolset for PCP User Guide

Variable argument '...' operator. Used in C library function `va_start()`. Returns the stack offset to the variable argument list.

`__exit`

```
void __exit( const unsigned long srpn );
```

To allow re-arbitration 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 `srpn` value is loaded in R6.SRPN. 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.SRPN 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.SRPN and the interrupt flag in the EXIT instruction is disabled (EXIT EC=0, ST=0, INT=0, EP=1, cc_UC).

The `__exit()` intrinsic function is kept simple as apposed to the EXIT instruction. The concept of the program flow in C (and any high level language), is that you have a routine that starts at the top and runs to the end (return), and then the initiative is past to the 'caller' side again. The flow can be interrupted by an interrupt. With the EXIT instructions however, other interrupts can be started (in fact anything is possible), enabling all kind of 'unwanted' flow.

`__exit_cpu`

```
void __exit_cpu( const unsigned long srpn );
```

To service TriCore CPU interrupts, a 'voluntary exiting' scheme is supported via the intrinsic `__exit_cpu()`. 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 TriCore CPU service request and the `srpn` value is loaded in R6.SRPN.

`__Exit`

```
void __Exit( int status );
```

Exit unconditionally. Same as C library function `_Exit()`. This intrinsic generates an EXIT instruction with the following settings: EC=0, ST=1, INT=0, EP=0, cc_UC. The CEN flag is cleared. Returns with `status` as the return value.

`__free`

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

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

`__get_return_address`

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

Returns the return address of a function.

__ien

```
void __ien( const unsigned int value );
```

Set or clear the IEN flag. You can use this intrinsic safely only when the R7 flags are preserved by the compiler (**--preserve-r7-flags**).

__ld32_fpi

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

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

Example:

```
#include <regtcl791.sfr>
unsigned int ld32( void )
{
    return __ld32_fpi( (unsigned long)&(P10_OUT.U) );
}
```

generates:

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

__nop

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

A NOP instruction is generated.

__pri

```
unsigned int __pri( unsigned int value );
```

Use `PRI R[b], R[a], cc_UC` instruction to prioritize value.

__rl

```
unsigned int __rl( unsigned int value,
                  unsigned int count );
```

Rotate `value` left `count` times. This intrinsic uses `RL R[a], Imm5` instruction(s). Returns the rotated value.

__rr

```
unsigned int __rr( unsigned int value,
                  unsigned int count );
```

Rotate `value` right `count` times. This intrinsic uses `RR R[a], Imm5` instruction(s). Returns the rotated value.

__st32_fpi

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

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

Example:

```
#include <regtcl791.sfr>
void st32( unsigned int value )
{
    __st32_fpi( (unsigned long)(P10_OUT.U), value );
}
```

generates:

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

__xchf8

```
unsigned int volatile __xchf8( unsigned int value,
                               unsigned int __far * address );
```

Exchange the 8-bit contents of register `value` and FPI variable. `address` must be a constant FPI address expression. This intrinsic generates a `XCH.F R[b], [R[a]]` instruction.

Returns: 8-bit contents of FPI address

__xchf16

```
unsigned int volatile __xchf16( unsigned int value,
                                unsigned int __far * address );
```

Exchange the 16-bit contents of register `value` and FPI variable. `address` must be a constant FPI address expression. This intrinsic generates a `XCH.F R[b], [R[a]]` instruction.

Returns: 16-bit contents of FPI address

__xchf32

```
unsigned int volatile __xchf32( unsigned int value,
                               unsigned int __far * address );
```

Exchange the 32-bit contents of register `value` and FPI variable. `address` must be a constant FPI address expression. This intrinsic generates a `XCH.F R[b], [R[a]]` instruction.

Returns: 32-bit contents of FPI address

__xchpi

```
unsigned int volatile __xchpi( unsigned int value,
                               unsigned int __near * address );
```

Exchange the 32-bit contents of register `value` and PRAM variable. `address` must be a constant PRAM address expression. This intrinsic generates a `XCH.PI R[a], [#offset6]` instruction.

Returns: 32-bit contents of PRAM address

1.9. PCP Code Generation

The effectiveness of the code generated by the PCP C compiler strongly depends on its stack implementation which is a static one. This means automatic stack variables, function stack parameters and temporary data are stored in overlayable static data areas.

A dynamic stack cannot be supported because the PCP instruction set does not have push and pop instructions. Simulating push and pop instructions is not an option because that requires registers by itself. This might force the compiler to abort in cases where a register must be pushed but all registers are in use.

A static stack poses the restriction that functions cannot be reentrant or recursive. Additionally, function pointer prototypes are limited to register parameters as the remaining parameters require a dynamic stack. Because the PCP does not have a hardware stack, function return addresses are stored on the static stack as well.

For effectiveness of the code generated by the PCP C compiler it is advised to use MIL linking and MIL archives, which is the default for building PCP applications with the Eclipse development environment.

The PCP core is designed to support high(est) priority non-interruptible functions most effectively. It is defined by Infineon to use a programming model for PCP applications that mainly consists of non-interruptible functions called *non-interruptible PCP channels*. The TriCore itself is intended to be used as the scheduler and arbitrator for these high priority non-interruptible PCP channels. Interruptible PCP channels, that can interrupt each other, are intended to be an exceptional case, they are supported but not by default.

1.9.1. Non-interruptible Code Generation

By default the compiler supports code generation for high(est) priority non-interruptible functions. Functions using a static stack are implicitly not interruptible, because they are not reentrant. The interrupt flag is disabled and must be kept disabled (IEN=0).

TASKING VX-toolset for PCP User Guide

Lower priority interruptible functions can only be supported when they do not share static stack space, which means that they cannot have common functions. See [Section 1.9.2, Interruptible Code Generation](#).

Preserving R7.IEN and R7.CEN when updating DPTR for PRAM access is superfluous if functions are not interruptible. For example, when accessing global variable "x" in PRAM the next code is generated.

```
ldl.il  r7,@DPTR(_PCP_x) ; load R7.DPTR, R7.[7..0] flags are cleared
st.pi  r5,[_PCP_x]
```

Not preserving the IEN and CEN reduces the amount of generated code substantially, an average of 45%¹ of the code size is saved. The CEN flag is not explicitly set for each PRAM access, although it would not cost any extra code for direct PRAM access in the above example, but for each indirect PRAM access an extra set bit instruction is required to keep the CEN enabled.

A non-interruptible function can voluntarily exit, by using an `__exit()` intrinsic function. The `__exit()` SRPN argument specifies the interrupt channel that needs to be serviced. This can be itself or another interrupt. If another interrupt is serviced this interrupt cannot use the same functions that use the static stack, because functions using a static stack are not reentrant. Ignoring this requirement will result in undefined run-time behavior. The same interrupt can be serviced without any restrictions.

The IEN flag in the channel context, generated by the compiler for `__interrupt()` qualified functions, is set to zero, because functions are not interruptible by default. The interrupt priority of the channel R6.CPPN is set to zero, because channels that do not allow interrupts do not need a interrupt priority value. Channels that have IEN disabled in the channel context are serviced if their channel number is higher than the priority of the currently running channel.

The CEN flag in the context channel is set. For each PRAM access CEN is cleared. CEN is re-set at interrupt function return or at voluntary exit when using the `__exit()` intrinsic.

Depending on the channels start mode the channel continues execution at the next PC or restarts. The channels start at context PC mode is the default of the PCP (CS.RCB=0), the channel continues on the next PC after a voluntary exit (EP=1). Voluntary exit and channel start at base is not supported, it may lead to undefined run-time behavior. See the [__exit\(\) intrinsic function](#).

1.9.2. Interruptible Code Generation

Interruptible functions can only be supported when they do not share static stack space, which means that they cannot have common functions that use the static stack. The static stack is not only used for user defined functions, but also for compiler run-time functions and C library functions.

Each interrupt channel must be linked separately when they have commonly used functions. The [linker option `--link-only`](#) is used to link a single channel without locating it. Several linked channels can be located with the linker to a single PCP application. Functions that are used in different interrupt channels are duplicated, their names need to be unique to avoid duplicate name conflicts. With the [compiler option `--symbol-prefix="name"`](#) you can prefix all global variables with *name*. You need to rebuild the libraries with the prefix that corresponds to the channel. Each channel needs its own prefixed library functions. It is not required to create C libraries for each channel when C libraries are linked in the compiler with the option `--mil`.

¹This average is statistically determined on a large number of test programs.

To share global variables between PCP channels the qualifier `__share` needs to be used on the (external) definition. E.g. `extern int __share variable`. `__share` variables get application scope instead of channel scope, using the `_lc` linker label prefix.

To share global functions between PCP channels the PCP can post channel requests to itself using the `__exit(SRPN)` intrinsic function. For servicing TriCore CPU interrupts the `__exit_cpu(SRPN)` intrinsic need to be used. The SRPN is the channel number to service, which can be an interruptible or non-interruptible channel.

The IEN flag in the channel context, generated by the compiler for `__interrupt()` qualified functions, is set to one, when interruptible code generation is enabled with the PCP [C compiler option `--interrupt-enable`](#).

The PCP interrupt priority CPPN can be set using `__cppn(CPPN)` interrupt function qualifier. The `__cppn()` function qualifier can only be used for `__interrupt()` qualified functions and have interrupting enabled. The R6.CPPN value is restored at function exit, because R6 is used by the PCP C compiler as general purpose register to store local objects.

Functions that are interruptible need to keep the IEN flag enabled or need to preserve its state. With the PCP [C compiler option `--interrupt-enable`](#) the IEN flag is kept enabled.

With PCP [C compiler option `--preserve-r7-flags`](#) code is generated for PRAM access that does not write any of the R7.0..7 flags. Preserving the R7 flags increases the generated code substantially, but the state of the IEN and CEN flags can be changed anywhere in the C code. For example the interrupts can be temporary turned off in the code to call a non-interruptible function and turn it on afterwards. Also the channel can be turned off to prevent servicing the channel anymore. The code generated for accessing global PRAM variable "x" is:

```
ld.i    r1,0x3f
ldl.il  r1,0xff
and     r7,r1,cc_uc
ldl.il  r1,@DPTR(_PCP_x)
or      r7,r1,cc_uc ; load R7.DPTR
st.pi   r5,[_PCP_x]
```

With the intrinsics `__ien()` and `__cen()` the IEN and CEN flags can be set or cleared in the code. These intrinsics can only be safely used when the r7 flags are preserved by the compiler.

When [C compiler option `--preserve-r7-flags`](#) is not used and [C compiler option `--interrupt-enable`](#) is used code is generated by the PCP C compiler that keeps the IEN and CEN flags enabled when accessing PCP PRAM. For example, when accessing global variable "x" in PRAM next code is generated.

```
ldl.ilf  r7,@DPTR_FLAGS(_PCP_x,0x60) ; load R7.DPTR, IEN=1 and CEN=1
st.pi    r5,[_PCP_x]
```

For direct PRAM access the generated code is comparable for interruptible and un-interruptible code generation ([C compiler option `--interrupt-enable`](#)). Indirect PRAM access is smaller and faster for un-interruptible code generation, because for interruptible code generation the IEN and CEN flags are set for every indirect PRAM access. For example, the next indirect PRAM access is generated for [C compiler option `--interrupt-enable`](#).

```
ldl.il  r3,0x3fc0
and     r3,r1,cc_uc
```

```
shl    r3,0x2
set    r3,0x5 ; set IEN
set    r3,0x6 ; set CEN
mov    r7,r3,cc_uc ; load R7.DPTR
st.p   r5,[r1],cc_uc
```

The generated code with **--interrupt-enable** averages 1.5% larger in code size and 1.3% slower in performance than the default code generation. With **--preserve-r7-flags** the generated code increases an average of 45%².

1.10. Compiler Generated Sections

The compiler generates code and data in several types of sections. The compiler uses the following section naming convention:

section_type_prefix.section_type

The default section name is equal to the generated section type that is prefixed with `.pcptext.` for code in CMEM and `.pcpdata.` for data in PRAM. The default code section name is `.pcptext.code` and the default PRAM data section name is `.pcpdata.data`. The default FPI data section name uses a prefix conform the TriCore C compiler. E.g. `__far int i;` is located in a section with name `.bss.linear`. You can change section names with `#pragma section` or with the command line option **--rename-sections**.

The following table lists the section types and name prefixes.

Section type	Name prefix	Description
code	<code>.pcptext</code>	program code
data	<code>.pcpdata</code>	initialized <code>__near</code> data
linear	<i>conform TriCore</i>	initialized <code>__far</code> data

The section names are independent of the section attributes such as `clear`, `init`, `max`, and `overlay`.

Section names are case sensitive. By default, the sections are not concatenated by the linker. This means that multiple sections with the same name may exist. At link time, sections with different attributes can be selected by their attributes. The linker may remove unreferenced sections from the application.

Overlay sections

For static stack overlay sections the compiler uses a different section naming convention. The section name equals the function name in which the overlay section is allocated.

function_name

For example, for function `main`, the section name will be:

`__PCP_main`

²This average is statistically determined on a large number of test programs.

1.10.1. Rename Sections

You can rename sections with a pragma or with a command line option. The syntax is the same:

```
--rename-sections=[type=]format_string[, [type=]format_string]...
```

```
#pragma section [type=]format_string[, [type=]format_string]...
```

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

Some examples (file `test.c`):

```
#pragma section data={module}_{type}_{attrib}
int x;
/* Section name: .pcpdata.test_data_data_clear */

#pragma section data=cpcp_{module}_{name}
int status;
/* Section name: .pcpdata.cpcp_test_status */

#pragma section data=RENAMED_{name}
int barcode;
/* Section name: .pcpdata.RENAMED_barcode */

section_type_prefix.module_name.pragma_value
```

#pragma endsection

With the `#pragma endsection` the default section name is restored. Nesting of pragma section/endsection pairs will save the status of the previous level.

Examples (file `example.c`)

```
char a;    // allocated in '.pcpdata.data'
#pragma section data=MyData1
char b;    // allocated in '.pcpdata.MyData1'
#pragma section data=MyData2
char c;    // allocated in '.pcpdata.MyData2'
#pragma endsection
char d;    // allocated in '.pcpdata.MyData1'
#pragma endsection
char e;    // allocated in '.pcpdata.data'
```


Chapter 2. Assembly Language

This chapter describes the most important aspects of the TASKING assembly language for the PCP. For a complete overview of the PCP2 architecture, refer to the *PCP2 Target Specification* [V 1.0, 2000-06, Infineon].

2.1. Assembly Syntax

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

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

The syntax of an assembly statement is:

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

label

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

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

Examples:

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

instruction

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

Operands are described in [Section 2.3, Operands of an Assembly Instruction](#). The instructions are described in the target's core Architecture Manual.

directive

With directives you can control the assembler from within the assembly source. Except for preprocessing directives, these must not start in the first column. Directives are described in [Section 2.9, Assembler Directives and Controls](#).

TASKING VX-toolset for PCP User Guide

macro_call A call to a previously defined macro. It must not start in the first column. See [Section 2.10, Macro Operations](#).

comment Comment, preceded by a ; (semicolon).

You can use empty lines or lines with only comments.

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

\$control

For more information on controls see [Section 2.9, Assembler Directives and Controls](#).

2.2. Assembler Significant Characters

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

Some characters have a special meaning to the assembler. Special characters associated with expression evaluation are described in [Section 2.6.3, Expression Operators](#). Other special assembler characters are:

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

2.3. Operands of an Assembly Instruction

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

Operand	Description
<i>symbol</i>	A symbolic name as described in Section 2.4, Symbol Names . Symbols can also occur in expressions.
<i>register</i>	Any valid register as listed in Section 2.5, Registers .
<i>expression</i>	Any valid expression as described in Section 2.6, Assembly Expressions .
<i>address</i>	A combination of <i>expression</i> , <i>register</i> and <i>symbol</i> .

Addressing modes

The PCP assembly language has several addressing modes. These addressing modes are used for FPI addressing, PRAM data indirect addressing or flow control destination addressing. For details see the *PCP2 Target Specification* [V 1.0, 2000-06, Infineon].

2.4. Symbol Names

User-defined symbols

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

Predefined preprocessor symbols

These symbols start and end with two underscore characters, `__symbol__`, and you can use them in your assembly source to create conditional assembly. See [Section 2.4.1, Predefined Preprocessor Symbols](#).

Labels

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

Reserved symbols

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

Examples

Valid symbol names:

```
loop_1
ENTRY
a_B_c
_aBC
```

Invalid symbol names:

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

2.4.1. Predefined Preprocessor Symbols

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

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

Example

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

2.5. Registers

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

```
R0 .. R7    (general purpose registers)
```

2.5.1. Special Function Registers

It is easy to access Special Function Registers (SFRs) that relate to peripherals from assembly. The SFRs are defined in a special function register definition file (*.def) as symbol names for use assembler. The assembler reads the SFR definition file as defined by the selected derivative with the command line option **--cpu (-C)**. SFRs are defined with **.EQU** directives.

For example (from `regtc1165.def`):

```
PC      .equ  0xFE08
```

Without an SFR file the assembler only knows the general purpose registers R0-R7.

2.6. Assembly Expressions

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

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

Expressions follow the conventional rules of algebra and boolean arithmetic.

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

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

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

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

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

All types of expressions are explained in separate sections.

2.6.1. Numeric Constants

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

Base	Description	Example
Binary	A 0b or 0B prefix followed by binary digits (0,1).	0B1101 0b11001010
Hexadecimal	A 0x or 0X prefix followed by hexadecimal digits (0-9, A-F, a-f).	0X12FF 0x45 0xfa10
Decimal integer	Decimal digits (0-9).	12 1245
Decimal floating-point	Decimal digits (0-9), includes a decimal point, or an 'E' or 'e' followed by the exponent.	6E10 .6 3.14 2.7e10

2.6.2. Strings

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

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

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

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

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

Examples

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

2.6.3. Expression Operators

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

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

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

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

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

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

2.7. Working with Sections

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

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

For more information about locating sections see [Section 5.7.8, The Section Layout Definition: Locating Sections](#).

Section definition

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

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

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

Section activation

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

```
.SECT "name"
```

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

Examples

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

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

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

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

2.8. Built-in Assembly Functions

The TASKING assembler has several built-in functions to support data conversion, string comparison, and math computations. You can use functions as terms in any expression.

Syntax of an assembly function

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

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

The names of assembly functions are case insensitive.

Overview of mathematical functions

Function	Description
@ABS (<i>expr</i>)	Absolute value
@ACS (<i>expr</i>)	Arc cosine
@ASN (<i>expr</i>)	Arc sine

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

Overview of conversion functions

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

Overview of string functions

Function	Description
@CAT (<i>str1</i> , <i>str2</i>)	Concatenate <i>str1</i> and <i>str2</i>

Function	Description
@LEN (<i>string</i>)	Length of string
@POS (<i>str1</i> , <i>str2</i> [, <i>start</i>])	Position of <i>str2</i> in <i>str1</i>
@SCP (<i>str1</i> , <i>str2</i>)	Compare <i>str1</i> with <i>str2</i>
@SUB (<i>str</i> , <i>expr1</i> , <i>expr2</i>)	Return substring

Overview of macro functions

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

Overview of address calculation functions

Function	Description
@DPTR (<i>expr</i>)	Returns bits 6-13 of the pcpdata address
@DPTRBIT (<i>expr</i>)	Returns single negated bit in the range 6-13 of the pcpdata address
@HI (<i>expr</i>)	Returns upper 16 bits of expression value
@INIT_R7 (<i>start</i> , <i>dptr</i> , <i>flags</i>)	Returns the 32-bit value to initialize R7
@LO (<i>expr</i>)	Returns lower 16 bits of expression value
@LSB (<i>expr</i>)	Least significant byte of the expression
@MSB (<i>expr</i>)	Most significant byte of the expression

Overview of assembler mode functions

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

Detailed Description of Built-in Assembly Functions

@ABS(*expression*)

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

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

Example:

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

@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 assembler executable. This is 'aspcp' for the PCP assembler.

Example:

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

@AT2(*expression1*,*expression2*)

Returns the arc tangent of *expression1*/*expression2* as a floating-point value in the range $-\pi$ to π . *expression1* and *expression2* must be separated by a comma.

Example:

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

@ATN(*expression*)

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

Example:

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

@CAT(*string1*,*string2*)

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

Example:

```
.DEFINE ID "@CAT('TASK','ING')" ;ID = 'TASKING'
```

@CEL(*expression*)

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

Example:

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

@CNT()

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

Example:

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

@COH(*expression*)

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

Example:

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

@COS(*expression*)

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

Example:

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

@CPU(*string*)

Returns integer 1 if *string* corresponds to the selected CPU type; 0 otherwise. See also [assembler option --cpu](#) (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 1 if *symbol* has been defined, 0 otherwise. *symbol* can be any symbol or label not associated with a `.MACRO` or `.SDECL` directive. If *symbol* is quoted, it is looked up as a `.DEFINE` symbol; if it is not quoted, it is looked up as an ordinary symbol or label.

Example:

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

@DPTR(*expression*)

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

Example:

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

@DPTRBIT(*expression*)

Returns a single negated bit in the range 6-13 of the pcdata address provided. This is equivalent to $((\text{expression} \gg n - 8 + 6) \wedge 0x1)$, where $n = 8..15$. The bit returned is defined by the BMOVN instruction.

Example:

```
bmovn r7,8,@DPTRBIT(label) ; bmovn R7,8,((label>>6)^0x1)
```

@EXP(*expression*)

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

Example:

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

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

@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*)

Returns the 32-bit fractional representation of the floating-point *expression*. The expression must be in the range $[-1,+1>$.

Example:

TASKING VX-toolset for PCP User Guide

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

@HI(*expression*)

Returns the upper 16 bits of a value. @HI(*expression*) is equivalent to $((\text{expression} \gg 16) \& 0xffff)$.

Example:

```
ldl.iu r5,#@HI(COUNT) ;upper 16 bits of COUNT
ldl.il r5,#@LO(COUNT)
```

@INIT_R7(*start,dptr,flags*)

Returns the 32-bit value needed to initialize R7. This is equivalent to $(\text{start} \ll 16) + (((\text{dptr} \& 0x3fff) \gg 6) \ll 8) + (\text{flags} \& 0xff)$.

Example:

```
.WORD @INIT_R7(start_0,pcp_data_0,7)
```

@INT(*expression*)

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

Example:

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

@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(*expression1,expression2*)

Concatenates the 16-bit *expression1* and *expression2* into a 32-bit word value such that *expression1* is the high half and *expression2* 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:

```
ldl.iu r5,#@HI(COUNT)
ldl.il r5,#@LO(COUNT) ;lower 16 bits of 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*. The result of the expression is calculated as 16 bit.

Example:

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

@LST()

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

Example:

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

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

Example:

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

@MAX(expression1[,expressionN],...)

Returns the maximum value of *expression1*, ..., *expressionN* as a floating-point value.

Example:

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

@MIN(expression1[,expressionN],...)

Returns the minimum value of *expression1*, ..., *expressionN* 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*. The result of the expression is calculated as 16 bit.

Example:

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

@MXP()

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

Example:

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

@POS(string1,string2[,start])

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

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

Example:


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

@POW(expression1,expression2)

Returns *expression1* raised to the power *expression2* as a floating-point value. *expression1* and *expression2* must be separated by a comma.

Example:

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

@RND()

Returns a random value in the range 0.0 to 1.0.

Example:

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

@RVB(expression1,expression2)

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

Example:

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

@SCP(string1,string2)

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

Example:

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

@SFRACT(expression)

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

Example:

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

@SGN(*expression*)

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

Example:

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

@SIN(*expression*)

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

Example:

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

@SNH(*expression*)

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

Example:

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

@SQT(*expression*)

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

Example:

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

@SUB(*string,expression1,expression2*)

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

Example:

```
.DEFINE ID "@SUB('TASKING',3,4)" ;ID = 'KING'
```

@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
```

2.9. Assembler Directives and Controls

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

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

The following directives fall under this group:

- Assembly control directives
- Symbol definition and section directives
- Data definition / Storage allocation directives
- High Level Language (HLL) directives
- Directives that are interpreted by the macro preprocessor. These directives tell the macro preprocessor how to manipulate your assembly code before it is actually being assembled. You can use these directives to write macros and to write conditional source code. Parts of the code that do not match the condition, will not be assembled at all.

TASKING VX-toolset for PCP User Guide

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

The following controls are available:

- Assembly listing controls
- Miscellaneous controls

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

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

2.9.1. Assembler Directives

Overview of assembly control directives

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

Overview of symbol definition and section directives

Directive	Description
<code>.ALIAS</code>	Create an alias for a symbol
<code>.EQU</code>	Set permanent value to a symbol
<code>.EXTERN</code>	Import global section symbol
<code>.GLOBAL</code>	Declare global section symbol
<code>.LOCAL</code>	Declare local section symbol
<code>.NAME</code>	Specify name of original C source file
<code>.ORG</code>	Initialize memory space and location counters to create a nameless section
<code>.SDECL</code>	Declare a section with name, type and attributes

Directive	Description
<code>.SECT</code>	Activate a declared section
<code>.SET</code>	Set temporary value to a symbol
<code>.SIZE</code>	Set size of symbol in the ELF symbol table
<code>.TYPE</code>	Set symbol type in the ELF symbol table
<code>.WEAK</code>	Mark a symbol as 'weak'

Overview of data definition / storage allocation directives

Directive	Description
<code>.ACCUM</code>	Define 64-bit constant of 18 + 46 bits format
<code>.ALIGN</code>	Align location counter
<code>.ASCII, .ASCIIZ</code>	Define ASCII string without / with ending NULL byte
<code>.BYTE</code>	Define byte
<code>.DOUBLE</code>	Define a 64-bit floating-point constant
<code>.FLOAT</code>	Define a 32-bit floating-point constant
<code>.FRACT</code>	Define a 32-bit constant fraction
<code>.HALF</code>	Define half-word (16 bits)
<code>.SFRACT</code>	Define a 16-bit constant fraction
<code>.SPACE</code>	Define storage
<code>.WORD</code>	Define word (32 bits)

Overview of macro preprocessor directives

Directive	Description
<code>.DEFINE</code>	Define substitution string
<code>.DUP, .ENDM</code>	Duplicate sequence of source lines
<code>.DUPA, .ENDM</code>	Duplicate sequence with arguments
<code>.DUPC, .ENDM</code>	Duplicate sequence with characters
<code>.DUPF, .ENDM</code>	Duplicate sequence in loop
<code>.IF, .ELIF, .ELSE</code>	Conditional assembly directive
<code>.ENDIF</code>	End of conditional assembly directive
<code>.EXITM</code>	Exit macro
<code>.MACRO, .ENDM</code>	Define macro
<code>.PMACRO</code>	Undefine (purge) macro
<code>.UNDEF</code>	Undefine <code>.DEFINE</code> symbol

Overview of HLL directives

Directive	Description
.CALLS	Pass call tree information and/or stack usage information
.MISRAC	Pass MISRA-C information

.ACCUM

Syntax

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

Description

With the `.ACCUM` directive 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

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

`.SPACE` (Define Storage)

.ALIAS

Syntax

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

Description

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

Example

```
exit .ALIAS _Exit
```


.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

```

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

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

```

.ASCII, .ASCIIZ

Syntax

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

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

Description

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

The `.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"
```

Note that 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

[.BYTE](#) (Define a constant byte)

[.SPACE](#) (Define Storage)

.BYTE

Syntax

```
[label:] .BYTE argument[,argument]...
```

Description

With the `.BYTE` directive 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).

String constants

Single-character strings are stored in a byte whose lower seven bits represent the ASCII value of the character, for example:

```
.BYTE 'R' ; = 0x52
```

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

```
.BYTE 'AB', , 'C' ; = 0x41420043 (second argument is empty)
```

Example

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

Related Information

[.ASCII](#), [.ASCIIZ](#) (Define ASCII string without/with ending NULL)

[.WORD](#), [.HALF](#) (Define a word / halfword)

[.SPACE](#) (Define Storage)

.CALLS

Syntax

```
.CALLS 'caller','callee'
```

or

```
.CALLS 'caller','','stack_usage[,...]
```

Description

The first syntax creates a call graph reference between *caller* and *callee*. The linker needs this information to build a call graph. *caller* and *callee* are names of functions.

The second syntax specifies stack information. When *callee* is an empty name, this means we define the stack usage of the function itself. The value specified is the stack usage in bytes at the time of the call including the return address. A function can use multiple stacks.

This information is used by the linker to compute the used stack within the application. The information is found in the generated linker map file within the Memory Usage.

This directive is generated by the C compiler. Normally you will not use it 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`.

```
.CALLS 'main','','8'
```

The function `main` uses 8 bytes on the stack.

.COMMENT

Syntax

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

Description

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

A label is not allowed before this directive.

Example

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

.DEFINE

Syntax

```
.DEFINE symbol string
```

Description

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

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

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

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

A label is not allowed before this directive.

Example

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

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

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

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

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

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

Related Information

`.UNDEF` (Undefine a `.DEFINE` symbol)

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

.DUP, .ENDM

Syntax

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

Description

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

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

Example

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

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

Related Information

- [.DUPA, .ENDM](#) (Duplicate sequence with arguments)
- [.DUPC, .ENDM](#) (Duplicate sequence with characters)
- [.DUPF, .ENDM](#) (Duplicate sequence in loop)
- [.MACRO, .ENDM](#) (Define a macro)

.DUPA, .ENDM

Syntax

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

Description

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

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

Example

Consider the following source input statements,

```
.DUPA VALUE,12,,32,34  
.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, .ENDM](#) (Duplicate sequence of source lines)
- [.DUPC, .ENDM](#) (Duplicate sequence with characters)
- [.DUPF, .ENDM](#) (Duplicate sequence in loop)
- [.MACRO, .ENDM](#) (Define a macro)

.DUPC, .ENDM

Syntax

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

Description

With the `.DUPC/.ENDM` directive you can repeat a block of source statements for each character within *string*. For each character in the string, the *formal_arg* parameter within the block is replaced with that character. If the string is empty, then the block is skipped.

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

Example

Consider the following source input statements,

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

This is expanded as follows:

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

Related Information

- `.DUP` , `.ENDM` (Duplicate sequence of source lines)
- `.DUPA` , `.ENDM` (Duplicate sequence with arguments)
- `.DUPF` , `.ENDM` (Duplicate sequence in loop)
- `.MACRO` , `.ENDM` (Define a macro)

.DUPF, .ENDM

Syntax

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

Description

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

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

Example

Consider the following source input statements,

```
.DUPF NUM,0,7  
.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` , `.ENDM` (Duplicate sequence of source lines)
- `.DUPA` , `.ENDM` (Duplicate sequence with arguments)
- `.DUPC` , `.ENDM` (Duplicate sequence with characters)
- `.MACRO` , `.ENDM` (Define a macro)

.END

Syntax

.END

Description

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

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

The assembler does not allow a label with this directive.

Example

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

Related Information

-

.EQU

Syntax

symbol **.EQU** *expression*

Description

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

Example

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

```
MYSYMBOL .EQU 0x4000
```

You cannot redefine the symbol `MYSYMBOL` after this.

Related Information

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

.EXITM

Syntax

```
.EXITM
```

Description

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

A label is not allowed before this directive.

Example

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

Related Information

- `.DUP`, `.ENDM` (Duplicate sequence of source lines)
- `.DUPA`, `.ENDM` (Duplicate sequence with arguments)
- `.DUPC`, `.ENDM` (Duplicate sequence with characters)
- `.DUPF`, `.ENDM` (Duplicate sequence in loop)
- `.MACRO`, `.ENDM` (Define a macro)

.EXTERN

Syntax

```
.EXTERN symbol[,symbol]...
```

Description

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

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

A label is not allowed with this directive.

Example

```
.EXTERN AA,CC,DD      ;defined elsewhere
.sdecl ".pcptext.code", code
.sect ".pcptext.code"
.
.
LD.I R5,AA          ; AA is used here
.
```

Related Information

[.GLOBAL](#) (Declare global section symbol)

[.LOCAL](#) (Declare local section symbol)

.FAIL

Syntax

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

Description

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

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

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

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

A label is not allowed with this directive.

Example

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

This results in the error:

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

Related Information

[.MESSAGE](#) (Programmer generated message)

[.WARNING](#) (Programmer generated warning)

.FLOAT, .DOUBLE

Syntax

```
[label:].FLOAT expression[,expression]...
```

```
[label:].DOUBLE expression[,expression]...
```

Description

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

An *expression* can be:

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

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

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.

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

Example

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

Related Information

`.SPACE` (Define Storage)

.FRACT, .SFRACT

Syntax

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

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

Description

With the `.FRACT` or `.SFRACT` directive the assembler allocates and initializes a 32-bit or 16-bit constant fraction in memory 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 expression 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

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

`.SPACE` (Define Storage)

.GLOBAL

Syntax

```
.GLOBAL symbol [, symbol] . . .
```

Description

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

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

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

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

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

The assembler does not allow a label with this directive.

Example

```

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

Related Information

[.EXTERN](#) (Import global section symbol)

[.LOCAL](#) (Declare local section symbol)

.IF, .ELIF, .ELSE, .ENDIF

Syntax

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

Description

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

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

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

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

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

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

A label is not allowed with this directive.

Example

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

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

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

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

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

You can also define the symbols on the command line with the [assembler option --define \(-D\)](#):

```
aspcp --define=DEMO --define=TEST=0 test.src
```

.INCLUDE

Syntax

```
.INCLUDE "filename" | <filename>
```

Description

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

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

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

1. The current directory if you use the `"filename"` construction.
The current directory is not searched if you use the `<filename>` syntax.
2. The path that is specified with the [assembler option `--include-directory`](#).
3. The path that is specified in the environment variable `ASPCPINC` when the product was installed.
4. The default `include` directory in the installation directory.

The assembler does not allow a label with this directive.

Example

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

.LOCAL

Syntax

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

Description

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

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

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

The assembler does not allow a label with this directive.

Example

```
.SDECL ".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](#) (Import global section symbol)

[.GLOBAL](#) (Declare global section symbol)

.MACRO, .ENDM

Syntax

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

Description

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

The definition of a macro consists of three parts:

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

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

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

You can use the following operators in macro definition statements:

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

Example

The macro definition:

```
CONST.D .MACRO reg,value ;header
        ldl.iu reg,@HI(value) ;body
```



```

        ldl.il  reg,@LO(value)
        .ENDM
;terminator

```

The macro call:

```

        .SDECL  ".pcptext.code",code
        .SECT   ".pcptext.code"
        CONST.D r5,0x12345678

```

The macro expands as follows:

```

        ldl.iu  r5,@HI(0x12345678)
        ldl.il  r5,@LO(0x12345678)

```

Related Information

Section 2.10, *Macro Operations*

- [.DUP](#), [.ENDM](#) (Duplicate sequence of source lines)
- [.DUPA](#), [.ENDM](#) (Duplicate sequence with arguments)
- [.DUPC](#), [.ENDM](#) (Duplicate sequence with characters)
- [.DUPF](#), [.ENDM](#) (Duplicate sequence in loop)
- [.PMACRO](#) (Undefine macro)
- [.DEFINE](#) (Define a substitution string)

.MESSAGE

Syntax

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

Description

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

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

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

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

A label is not allowed with this directive.

Example

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

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

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

Related Information

[.FAIL](#) (Programmer generated error)

[.WARNING](#) (Programmer generated warning)

.MISRAC

Syntax

```
.MISRAC string
```

Description

The C compiler can generate the `.MISRAC` directive to pass the compiler's MISRA-C settings to the object file. The linker performs checks on these settings and can generate a report. It is not recommended to use this directive in hand-coded assembly.

Example

```
.MISRAC 'MISRA-C:2004,64,e2,0b,e,e11,27,6,ef83,e1,  
ef,66,cb75,af1,eff,e7,e7f,8d,63,87ff7,6ff3,4'
```

Related Information

[Section 3.7.2, C Code Checking: MISRA-C](#)

C compiler option `--misrac`

.NAME

Syntax

```
.NAME string
```

Description

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

Example

```
.NAME "main.c"
```

.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:

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

For more information about the section types and attributes see the [assembler directive .SDECL](#).

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

Example

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

; define a relocatable nameless section
.org

; define a relocatable data section
.org ,data

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

Related Information

[.SDECL](#) (Declare section name and attributes)

[.SECT](#) (Activate a declared section)

.PMACRO

Syntax

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

Description

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

The assembler does not allow a label with this directive.

Example

```
.PMACRO MAC1,MAC2
```

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

Related Information

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

.SDECL

Syntax

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

Description

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

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

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

The section type and attributes are case insensitive.

The defined *attributes* are:

Attribute	Description	Allowed on type
AT <i>address</i>	Locate the section at the given <i>address</i> .	CODE, DATA
CLEAR	Sections are zeroed at startup.	DATA
CLUSTER(' <i>name</i> ')	Cluster code sections with companion debug sections. Used by the linker during removal of unreferenced sections. The name must be unique for this module (not for the application).	CODE, DATA, DEBUG
GROUP(' <i>group</i> ')	Used to group sections.	DATA
INIT	Defines that the section contains initialization data, which is copied from ROM to RAM at program startup.	CODE, DATA
LINEAR	Section in the FPI space (TriCore linear address space).	DATA
MAX	When data sections with the same name occur in different object modules with the MAX attribute, the linker generates a section of which the size is the maximum of the sizes in the individual object modules.	DATA
NOCLEAR	Sections are not zeroed at startup. This is a default attribute for data sections. This attribute is only useful with BSS sections, which are cleared at startup by default.	DATA
NOINIT	Defines that the section contains no initialization data.	CODE, DATA
NOREAD	Defines that the section can be executed from but not read.	CODE
OVERLAY(' <i>name</i> ')	Static stack overlay. Automatic stack variables, function stack parameters and temporary data are stored here.	DATA

Attribute	Description	Allowed on type
PROTECT	Tells the linker to exclude a section from unreferenced section removal and duplicate section removal.	CODE, DATA
ROM	Section contains data to be placed in ROM. This ROM area is not executable.	CODE, DATA

Section names

The *name* of a section can have a special meaning for locating sections. The name of code sections should always start with ".pcptext". The name of data sections in PRAM should always start with ".pcpdata". With data sections in FPI space (data, linear), the prefix in the name is important. The prefix determines if the section is initialized, constant or uninitialized and which addressing mode is used. See the following table.

Section name for linear data	Type of section
.data.linear	initialized __far data
.rodata.linear	constant __far data
.bss.linear	uninitialized __far data

Note that the compiler uses the following name convention by default:

prefix.space

where *space* can be code, data or linear: In the C language you can overrule the default section name with `#pragma section`.

For static stack overlay sections the compiler uses a different section naming convention. The section name equals the function name in which the overlay section is allocated.

Group names

The GROUP attribute results in an extended section name. The name resulting from the .SDECL directive is as follows:

section-name[@group]

For example:

```
.sdecl ".pcpdata.data",data,group('groupname')
```

results in a section named: .pcpdata.data@groupname.

Example

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

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



```
.sdecl  '_PCP_main', data, overlay('stack_data')  
                                     ; declare overlay section  
.sect   '_PCP_main'                   ; activate section  
  
.sdecl  ".pcpdata.abssec", data at 0x100  
                                     ; absolute section  
.sect   ".pcpdata.abssec"             ; activate section
```

Related Information

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

Example

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

Related Information

`.SDECL` (Declare section name 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 the `.SET` directive again. Symbols that you define with the `.SET` directive are always local: you cannot define the symbol global with the `.GLOBAL` directive.

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

Example

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

Related Information

[.EQU](#) (Set permanent value to a symbol)

.SIZE

Syntax

```
.SIZE symbol,expression
```

Description

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

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

Example

```
_PCP_str: .type    object    ; object _PCP_str  
        .size    _PCP_str,4  ; size of object  
        .word    80  
        .word    67  
        .word    80  
        .word    0
```

Related Information

`.TYPE` (Set symbol type)

.SPACE

Syntax

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

Description

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

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

The *expression* specifies the number of MAUs (Minimal Addressable Units) to be reserved, and how much the location counter will advance. The expression must evaluate to an integer greater than zero and cannot contain any forward references (symbols that have not yet been defined). For the TriCore the MAU size is 8 (1 byte).

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 PRAM data section:

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

Related Information

[.BYTE](#) (Define a constant byte)

.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

```
_PCP_Afunc: .type func
```

Related Information

[.SIZE](#) (Set symbol size)

.UNDEF

Syntax

```
.UNDEF symbol
```

Description

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

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

The assembler does not allow a label with this directive.

Example

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

```
.UNDEF LEN
```

Related Information

[.DEFINE](#) (Define a substitution string)

.WARNING

Syntax

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

Description

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

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

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

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

A label is not allowed with this directive.

Example

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

This results in the warning:

```
w144: ["filename" line] Parameter out of range
```

Related Information

[.FAIL](#) (Programmer generated error)

[.MESSAGE](#) (Programmer generated message)

.WEAK

Syntax

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

Description

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

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

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

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

Example

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

Related Information

[.EXTERN](#) (Import global section symbol)

[.GLOBAL](#) (Declare global section symbol)

.WORD, .HALF

Syntax

```
[label:] .WORD argument[,argument]...  
[label:] .HALF argument[,argument]...
```

Description

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

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

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

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

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

String constants

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

```
.WORD 'R'           ; = 0x52000000  
.HALF 'R'          ; = 0x5200
```

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

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

Example

When a string is supplied as argument of a directive that initializes multiple bytes, each character in the string is stored in consecutive bytes whose lower seven bits represent the ASCII value of the character. For example:

```
HTBL: .HALF 'ABC',,'D'   ; results in 0x424100004400 , the 'C' is truncated  
WTBL: .WORD 'ABC'       ; results in 0x43424100
```

Related Information

[.BYTE](#) (Define a constant byte)

[.SPACE](#) (Define Storage)

2.9.2. Assembler Controls

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

Overview of assembler listing controls

Control	Description
<code>\$LIST ON/OFF</code>	Print / do not print source lines to list file
<code>\$LIST "flags"</code>	Exclude / include lines in assembly list file
<code>\$PAGE</code>	Generate form feed in list file
<code>\$PAGE settings</code>	Define page layout for assembly list file
<code>\$PRCTL</code>	Send control string to printer
<code>\$STITLE</code>	Set program subtitle in header of assembly list file
<code>\$TITLE</code>	Set program title in header of assembly list file

Overview of miscellaneous assembler controls

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

\$CASE

Syntax

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

Default

```
$CASE ON
```

Description

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

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

Example

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

Related Information

Assembler option **`--case-insensitive`**

\$DEBUG

Syntax

```
$DEBUG ON
$DEBUG OFF
$DEBUG "flags"
```

Default

```
$DEBUG "AhLS"
```

Description

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

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

a/A	Assembly source line information
h/H	Pass high level language debug information (HLL)
l/L	Assembler local symbols debug information
s/S	Smart debug information

You cannot specify `$DEBUG "ah"`. Either the assembler generates assembly source line information, or it passes HLL debug information.

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 Information

Assembler option `--debug-info`

\$HW_ONLY

Syntax

`$HW_ONLY`

Description

Normally the assembler replaces instructions by other, smaller or faster instructions.

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 Information

Assembler option `--optimize=+generics`

\$IDENT

Syntax

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

Default

```
$IDENT LOCAL
```

Description

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

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

Example

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

Related Information

Assembler directive [.GLOBAL](#)

Assembler directive [.LOCAL](#)

Assembler option [--symbol-scope](#)

\$LIST ON/OFF

Syntax

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

Default

```
$LIST ON
```

Description

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

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

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

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

Example

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

Related Information

Assembler option **--list-file**

Assembler control **\$LIST "flags"**

Assembler function **@LST()**

\$LIST "flags"

Syntax

```
$LIST "flags"
```

You can set the following flags:

d/D	List section directives (.SDECL, .SECT)
e/E	List symbol definition directives
g/G	List expansion of generic instructions
i/I	List generic instructions
m/M	List macro definitions
n/N	List empty source lines (newline)
p/P	List conditional assembly
q/Q	List equate and set directives (.EQU, .SET)
r/R	List relocations characters 'r'
v/V	List equate and set values
w/W	Wrap source lines
x/X	List macro expansions
y/Y	List cycle counts
z/Z	List define expansions

Default

```
$LIST "dEGiMnPqrVwXyZ"
```

Description

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

To switch a flag 'on', use a lowercase letter. To switch a flag off, use an uppercase letter.

Example

The following example also includes macro definitions and equate and set values in the list file:

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

Related Information

Assembler option **--list-file**

Assembler control **\$LIST ON/OFF**

Assembler option **--list-format**

\$OBJECT

Syntax

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

Default

```
$OBJECT
```

Description

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

Example

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

Related Information

Assembler option `--output`

\$PAGE

Syntax

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

Default

```
$PAGE 132,72,0,0,0
```

Description

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

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

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

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

Example

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

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

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

Related Information

Assembler option `--list-file`

\$PRCTL

Syntax

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

Description

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

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

You can specify the following arguments:

- expr* A byte expression which may be used to encode non-printing control characters, such as ESC.
- string* An assembler string, which may be of arbitrary length, up to the maximum assembler-defined limits.

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

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

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

Example

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

Related Information

[Assembler option --list-file](#)

\$STITLE

Syntax

```
$STITLE "string"
```

Default

```
$STITLE ""
```

Description

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

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

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

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

Example

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

Related Information

Assembler option **--list-file**

Assembler control **\$TITLE**

\$TITLE

Syntax

```
$TITLE "string"
```

Default

```
$TITLE ""
```

Description

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

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

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

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

Example

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

Related Information

[Assembler option --list-file](#)

[Assembler control \\$STITLE](#)

\$WARNING OFF

Syntax

```
$WARNING OFF [number]
```

Default

All warnings are reported.

Description

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

Example

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

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

Related Information

Assembler option **--no-warnings**

2.10. Macro Operations

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

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

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

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

2.10.1. Defining a Macro

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

The definition of a macro consists of three parts:

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

A macro definition takes the following form:

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

For more information on the definition see the description of the [.MACRO directive](#).

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

2.10.2. Calling a Macro

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

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

where,

TASKING VX-toolset for PCP User Guide

<i>label</i>	An optional label that corresponds to the value of the location counter at the start of the macro expansion.
<i>macro_name</i>	The name of the macro. This may not start in the first column.
<i>argument</i>	One or more optional, substitutable arguments. Multiple arguments must be separated by commas.
<i>comment</i>	An optional comment.

The following applies to macro arguments:

- Each argument must correspond one-to-one with the formal arguments of the macro definition. If the macro call does not contain the same number of arguments as the macro definition, the assembler issues a warning.
- If an argument has an embedded comma or space, you must surround the argument by single quotes (`'`).
- You can declare a macro call argument as null in three ways:

- enter delimiting commas in succession with no intervening spaces

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

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

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

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

2.10.3. Using Operators for Macro Arguments

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

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

Example: Argument Concatenation Operator - \

Consider the following macro definition:

```
SWAP_MEM .MACRO REG1,REG2           ;swap memory contents
    LD.P R4,[R\REG1],CC_UC         ;use R4 as temp
    LD.P R5,[R\REG2],CC_UC         ;use R5 as temp
    ST.P R5,[R\REG1],CC_UC
    ST.P R4,[R\REG2],CC_UC
.ENDM
```

The macro is called as follows:

```
SWAP_MEM 0,1
```

The macro expands as follows:

```
LD.P R4,[R0],CC_UC
LD.P R5,[R1],CC_UC
ST.P R5,[R0],CC_UC
ST.P R4,[R1],CC_UC
```

The macro preprocessor substitutes the character '0' for the argument `REG1`, and the character '1' for the argument `REG2`. The concatenation operator (`\`) indicates to the macro preprocessor that the substitution characters for the arguments are to be concatenated with the character 'A'.

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

```
LD.P R4,[RREG1],CC_UC
LD.P R5,[RREG2],CC_UC
ST.P R5,[RREG1],CC_UC
ST.P R4,[RREG2],CC_UC
```

which results in an assembler error (invalid operand).

Example: Decimal Value Operator - ?

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

Consider the following source code that calls the macro `SWAP_SYM` after the argument `AREG` has been set to 0 and `BREG` has been set to 1.

```
AREG .SET      0
BREG .SET      1
SWAP_SYM AREG,BREG
```

If you want to replace the arguments with the value of `AREG` and `BREG` rather than with the literal strings 'AREG' and 'BREG', you can use the `?` operator and modify the macro as follows:

```
SWAP_SYM .MACRO REG1,REG2           ;swap memory contents
    LD.P R4,[R\?REG1],CC_UC         ;use R4 as temp
```

TASKING VX-toolset for PCP User Guide

```
LD.P R5,[R\?REG2],CC_UC ;use R5 as temp
ST.P R5,[R\?REG1],CC_UC
ST.P R4,[R\?REG2],CC_UC
.ENDM
```

The macro first expands as follows:

```
LD.P R4,[R\?AREG],CC_UC
LD.P R5,[R\?BREG],CC_UC
ST.P R5,[R\?AREG],CC_UC
ST.P R4,[R\?BREG],CC_UC
```

Then ?AREG is replaced by '0' and ?BREG is replaced by '1':

```
LD.P R4,[R\1],CC_UC
LD.P R5,[R\2],CC_UC
ST.P R5,[R\1],CC_UC
ST.P R4,[R\2],CC_UC
```

Because of the concatenation operator '\' the strings are concatenated:

```
LD.P R4,[R1],CC_UC
LD.P R5,[R2],CC_UC
ST.P R5,[R1],CC_UC
ST.P R4,[R2],CC_UC
```

Example: Hex Value Operator - %

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

Consider the following macro definition:

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

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

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

The macro expands as follows:

```
HEXA NOP
```

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

Example: Argument String Operator - "

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

Consider the following macro definition:

```
STR_MAC      .MACRO  STRING
              .BYTE  "STRING"
              .ENDM
```

The macro is called as follows:

```
STR_MAC  ABCD
```

The macro expands as follows:

```
.BYTE  'ABCD'
```

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

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

If the macro is called as follows:

```
STR_MAC  sentence
```

it expands as:

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

Macro Local Label Override Operator - ^

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

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

Consider the following macro definition:

```
INIT .MACRO  ARG, CNT
      LD.I  R5, 0x1
^LAB:
      .WORD ARG
      ADD.I R5, 0x1
      COMP.I R5, #CNT
```

TASKING VX-toolset for PCP User Guide

```
JC ^LAB,CC_NZ
.ENDM
```

The macro is called as follows:

```
INIT 1,2
```

The macro expands as:

```
LD.I    R5,0x1
LAB:
.WORD 2
ADD.I   R5,0x1
COMP.I  R5,#4
JC      LAB,CC_NZ
```

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

```
LD.I    R5,0x1
LAB__M_L000001:
.WORD 2
ADD.I   R5,0x1
COMP.I  R5,#4
JC      LAB__M_L000001,CC_NZ
```

2.11. Generic Instructions

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

Generic jump JG

The PCP C compiler only generates generic direct jumps. Generic jump instructions are optimized by the PCP assembler to the real jump instructions depending on the operands. The PCP assembler supports the following generic jump:

```
jg _label [,cc_X] ; Jump Generic to label if condition cc_X is true
```

This generic jump is translated to:

- JC -> If the target address fits within the relative range of +/- 32 instructions.
- JL -> If condition code is `cc_UC` and the target address fits within the relative range of +/- 512 instructions.
- JC.A -> If the target address does not fit within the relative range.

If a condition code is omitted, the `cc_UC` condition code is used.

The indirect jumps JC.I and JC.IA are directly generated by the PCP compiler. Indirect jumps cannot be optimized by PCP assembler.

Generic bit handling instruction

```
bmovn R[a],#imm5,#imm1
```

This instruction moves the negated bit to a single bit in R[a]. If imm1 is 0 a set single bit in R[a] is done base on imm5. If imm1 is 1 a clr single bit in R[a] is done based on imm5. (imm5 == [8..15]).

For example:

```
bmovn R7,8,@DPTRBIT(label) ; bmovn R7,8,((label>>6)^0x1)
```

Generic load 10-bit immediate long instruction

```
LDL.IIL R[a], #imm8, #imm2
```

This instruction loads the long 10-bit immediate data following into the lower 16-bit of R[a]. The imm8 is loaded into most significant 8-bits of the lower 16-bit of R[a] (R[a].8-R[a].15). The imm2 is loaded at bit offset 5 and 6 of R[a]. The bits in the lower 8-bit of R[a] are cleared. The most significant 16-bits of R[a] are unaffected. This generic instruction is compiled by the assembler to a LDL.IIL PCP instruction.

Example:

```
ldl.iil r7,@DPTR(label),0x3
```

This loads the page number of label in R7.DPTR and sets R7.IEN (R7.5) and R7.CEN (R7.6) to 1, bit 7 and bit 0-4 are cleared. The most significant 16-bits of R7 are unaffected.

Chapter 3. Using the C Compiler

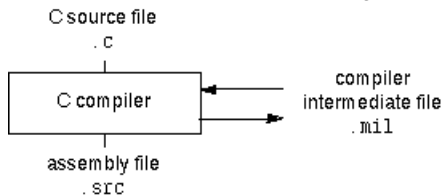
This chapter describes the compilation process and explains how to call the C compiler.

The TASKING VX-toolset for PCP under Eclipse can use the internal builder (default) or the TASKING makefile generator (external builder) to build your entire embedded project, from C source till the final ELF/DWARF object file which serves as input for the debugger.

Although in Eclipse you cannot run the C compiler separately from the other tools, this section discusses the options that you can specify for the C compiler.

On the command line it is possible to call the C compiler separately from the other tools. However, it is recommended to use the control program for command line invocations of the toolset (see [Section 6.1, Control Program](#)). With the control program it is possible to call the entire toolset with only one command line.

The C compiler takes the following files for input and output:



This chapter first describes the compilation process which consists of a *frontend* and a *backend* part. Next it is described how to call the C compiler and how to use its options. An extensive list of all options and their descriptions is included in [Section 8.1, C Compiler Options](#). Finally, a few important basic tasks are described, such as including the C startup code and performing various optimizations.

3.1. Compilation Process

During the compilation of a C program, the C compiler runs through a number of phases that are divided into two parts: *frontend* and *backend*.

The backend part is not called for each C statement, but starts after a complete C module or set of modules has been processed by the frontend (in memory). This allows better optimization.

The C compiler requires only one pass over the input file which results in relative fast compilation.

Frontend phases

1. The preprocessor phase:

The preprocessor includes files and substitutes macros by C source. It uses only string manipulations on the C source. The syntax for the preprocessor is independent of the C syntax but is also described in the ISO/IEC 9899:1999(E) standard.

2. The scanner phase:

TASKING VX-toolset for PCP User Guide

The scanner converts the preprocessor output to a stream of tokens.

3. The parser phase:

The tokens are fed to a parser for the C grammar. The parser performs a syntactic and semantic analysis of the program, and generates an intermediate representation of the program. This code is called MIL (Medium level Intermediate Language).

4. The frontend optimization phase:

Target processor independent optimizations are performed by transforming the intermediate code.

Backend phases

1. Instruction selector phase:

This phase reads the MIL input and translates it into Low level Intermediate Language (LIL). The LIL objects correspond to a processor instruction, with an opcode, operands and information used within the C compiler.

2. Peephole optimizer/instruction scheduler/software pipelining phase:

This phase replaces instruction sequences by equivalent but faster and/or shorter sequences, rearranges instructions and deletes unnecessary instructions.

3. Register allocator phase:

This phase chooses a physical register to use for each virtual register.

4. The backend optimization phase:

Performs target processor independent and dependent optimizations which operate on the Low level Intermediate Language.

5. The code generation/formatter phase:

This phase reads through the LIL operations to generate assembly language output.

3.2. Calling the C Compiler

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

Building a project under Eclipse

You have several ways of building your project:

- Build Selected File(s) (🔍). This compiles and assembles the selected file(s) without calling the linker.
 1. In the C/C++ Projects view, select the files you want to compile.

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

- Build Individual Project (🔧).

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

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

1. Select **Project » Clean...**

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

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

This way of building is not recommended for C/C++ development, but to enable this feature select **Project » Build Automatically** and ensure there is a check mark beside the **Build Automatically** menu item.

Select a target processor (core)

Processor options affect the invocation of all tools in the toolset. In Eclipse you only need to set them once. Based on the target processor, the compiler includes a special function register file. This is a regular include file which enables you to use virtual registers that are located in memory.

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

The Properties dialog appears.

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

In the right pane the Processor page appears.

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

To access the C compiler options

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

The Properties dialog appears.

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

In the right pane the Settings appear.

3. On the Tool Settings tab, select **C Compiler**.

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

Note that the C compiler options are used to create an object file from a C file. The options you enter in the Assembler page are not only used for hand-coded assembly files, but also for intermediate assembly files.

You can find a detailed description of all C compiler options in [Section 8.1, C Compiler Options](#).

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

```
cpcp [ [option]... [file]... ]...
```

3.3. The C Startup Code

You need the startup code to build an executable application. 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 TriCore C startup initializes and clears all global data as required, initializes the PCP compiler stack pointer, PRAM data page pointer and PCP status and control registers for each PCP interrupt function.

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()` function after finishing execution.

The PCP C startup code is part of the library and needs no further configuration.

For details on how to add or change the TriCore C startup code to your TriCore project, see the equivalent section in the *TASKING VX-toolset for TriCore User Guide*.

3.4. How the Compiler Searches Include Files

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

1. If the `#include` statement contains an absolute pathname, the compiler looks for this file. If no path or a relative path is specified, the compiler looks in the same directory as the source file. This is only possible for include files that are enclosed in "".

This first step is not done for include files enclosed in <>.

2. When the compiler did not find the include file, it looks in the directories that are specified in the **C Compiler » Include Paths** page in the **C/C++ Build » Settings » Tool Settings** tab of the Project Properties dialog (equivalent to the `-I` command line option).
3. When the compiler did not find the include file (because it is not in the specified include directory or because no directory is specified), it looks in the path(s) specified in the environment variable `CPCPINC`.

4. When the compiler still did not find the include file, it finally tries the default include directory relative to the installation directory (unless you specified [option `--no-stdinc`](#)).

Example

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

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

You can call the compiler as follows:

```
cpcp -Imyinclude test.c
```

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

3.5. Compiling for Debugging

Compiling your files is the first step to get your application ready to run on a target. However, during development of your application you first may want to debug your application.

To create an object file that can be used for debugging, you must instruct the compiler to include symbolic debug information in the source file.

To include symbolic debug information

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

The Properties dialog appears.

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

In the right pane the Settings appear.

3. On the Tool Settings tab, select **C Compiler » Debugging**.
4. Select **Default** in the **Generate symbolic debug information** box.

Debug and optimizations

Due to different compiler optimizations, it might be possible that certain debug information is optimized away. Therefore, if you encounter strange behavior during debugging it might be necessary to reduce the optimization level, so that the source code is still suitable for debugging. For more information on optimization see [Section 3.6, Compiler Optimizations](#).

Invocation syntax on the command line (Windows Command Prompt)

The invocation syntax on the command line is:

```
cpcp -g file.c
```

3.6. Compiler Optimizations

The compiler has a number of optimizations which you can enable or disable.

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

The Properties dialog appears.

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

In the right pane the Settings appear.

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

4. Select an optimization level in the **Optimization level** box.

or:

In the **Optimization level** box select **Custom optimization** and enable the optimizations you want on the Custom optimization page.

Optimization levels

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

- **Level 0 - No optimization:** No optimizations are performed. The compiler tries to achieve a 1-to-1 resemblance between source code and produced code. Expressions are evaluated in the order written in the source code, associative and commutative properties are not used.
- **Level 1 - Optimize:** Enables optimizations that do not affect the debug-ability of the source code. Use this level when you encounter problems during debugging your source code with optimization level 2.
- **Level 2 - Optimize more (default):** Enables more optimizations to reduce the memory footprint and/or execution time. This is the default optimization level.
- **Level 3 - Optimize most:** This is the highest optimization level. Use this level when your program/hardware has become too slow to meet your real-time requirements.
- **Custom optimization:** you can enable/disable specific optimizations on the Custom optimization page.

Optimization pragmas

If you specify a certain optimization, all code in the module is subject to that optimization. Within the C source file you can overrule the C compiler options for optimizations with `#pragma optimize flag` and `#pragma endoptimize`. Nesting is allowed:

```

#pragma optimize e      /* Enable expression
...                    simplification          */
... C source ...
...
#pragma optimize c      /* Enable common expression
...                    elimination. Expression
... C source ...      simplification still enabled */
...
#pragma endoptimize    /* Disable common expression
...                    elimination          */
#pragma endoptimize    /* Disable expression
...                    simplification      */

```

The compiler optimizes the code between the pragma pair as specified.

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

3.6.1. Generic Optimizations (frontend)

Common subexpression elimination (CSE) (option `-Oc/-OC`)

The compiler detects repeated use of the same (sub-)expression. Such a "common" expression is replaced by a variable that is initialized with the value of the expression to avoid recomputation. This method is called common subexpression elimination (CSE).

Expression simplification (option `-Oe/-OE`)

Multiplication by 0 or 1 and additions or subtractions of 0 are removed. Such useless expressions may be introduced by macros or by the compiler itself (for example, array subscription).

Constant propagation (option `-Op/-OP`)

A variable with a known value is replaced by that value.

Automatic function inlining (option `-Oi/-OI`)

Small functions that are not too often called, are inlined. This reduces execution time at the cost of code size.

Control flow simplification (option `-Of/-OF`)

A number of techniques to simplify the flow of the program by removing unnecessary code and reducing the number of jumps. For example:

- *Switch optimization*: A number of optimizations of a switch statement are performed, such as removing redundant case labels or even removing an entire switch.
- *Jump chaining*: A (conditional) jump to a label which is immediately followed by an unconditional jump may be replaced by a jump to the destination label of the second jump. This optimization speeds up execution.

TASKING VX-toolset for PCP User Guide

- *Conditional jump reversal:* A conditional jump over an unconditional jump is transformed into one conditional jump with the jump condition reversed. This reduces both the code size and the execution time.
- *Dead code elimination:* Code that is never reached, is removed. The compiler generates a warning messages because this may indicate a coding error.

Subscript strength reduction (option -Os/-OS)

An array or pointer subscripted with a loop iterator variable (or a simple linear function of the iterator variable), is replaced by the dereference of a pointer that is updated whenever the iterator is updated.

Loop transformations (option -Ol/-OL)

Transform a loop with the entry point at the bottom, to a loop with the entry point at the top. This enables constant propagation in the initial loop test and code motion of loop invariant code by the CSE optimization.

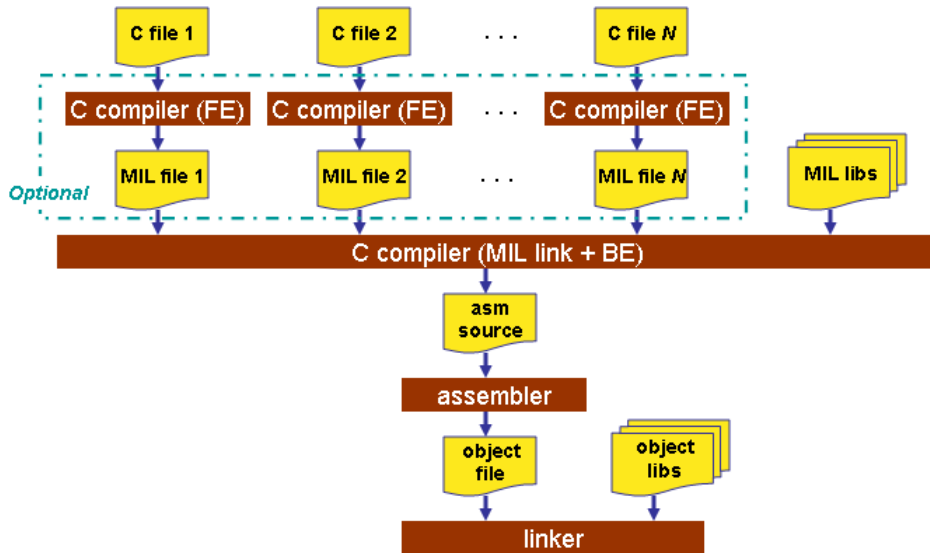
Forward store (option -Oo/-OO)

A temporary variable is used to cache multiple assignments (stores) to the same non-automatic variable.

MIL linking (Control program option --mil-link)

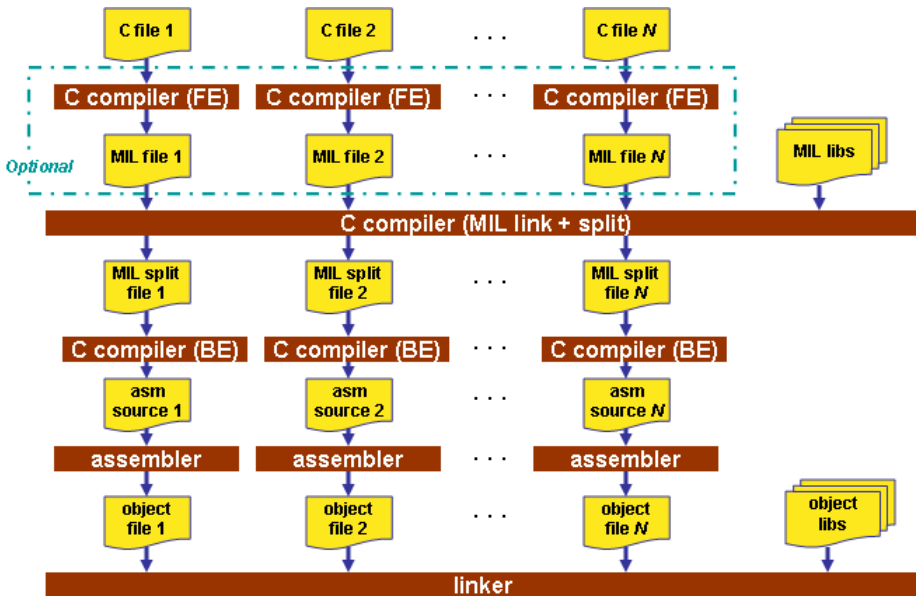
The frontend phase performs its optimizations on the MIL code. When all C modules and/or MIL modules of an application are given to the C compiler in a single invocation, the C compiler will link MIL code of the modules to a complete application automatically. Next, the frontend will run its optimizations again with application scope. After this, the MIL code is passed on to the backend, which will generate a single `.src` file for the whole application. Linking with the run-time library, floating-point library and C library is still necessary. Linking with the C library is required because this library contains some hand-coded assembly functions, that are not linked in at MIL level.

In the ISO C99 standard a "translation unit" is a preprocessed source file together with all the headers and source files included via the preprocessing directive `#include`. After MIL linking the compiler will treat the linked sources files as a single translation unit, allowing global optimizations to be performed, that otherwise would be limited to a single module.



MIL splitting (option `--mil-split`)

When you specify that the C compiler has to use MIL splitting, the C compiler will first link the application at MIL level as described above. However, after rerunning the optimizations the MIL code is not passed on to the backend. Instead the frontend writes a `.ms` file for each input module. A `.ms` file has the same format as a `.mil` file. Only `.ms` files that really change are updated. The advantage of this approach is that it is possible to use the `make` utility to translate only those parts of the application to a `.src` file that really have changed. MIL splitting is therefore a more efficient build process than MIL linking. The penalty for this is that the code compaction optimization in the backend does not have application scope. As with MIL linking, it is still required to link with the normal libraries to build an ELF file.



3.6.2. Core Specific Optimizations (backend)

Coalescer (option -Oa/-OA)

The coalescer seeks for possibilities to reduce the number of moves (MOV instruction) by smart use of registers. This optimizes both speed as code size.

Interprocedural register optimization (option -Ob/-OB)

Register allocation is improved by taking note of register usage in functions called by a given function.

Peephole optimizations (option -Oy/-OY)

The generated assembly code is improved by replacing instruction sequences by equivalent but faster and/or shorter sequences, or by deleting unnecessary instructions.

Code compaction (reverse inlining) (option -Or/-OR)

Compaction is the opposite of inlining functions: large chunks of code that occur more than once, are transformed into a function. This reduces code size at the cost of execution speed.

Generic assembly optimizations (option -Og/-OG)

A set of target independent optimizations that increase speed and decrease code size.

Automatic memory partitioning (option --no-partition)

The PCP has 256 pages of memory. Global variables are accessed by means of a page pointer (DPTR). With this optimization the compiler tries to allocate the global variables together in a page to reduce the

loading of the page pointer. So, this optimization reduces code size. This optimization is enabled by default.

3.6.3. Optimize for Size or Speed

You can tell the compiler to focus on execution speed or code size during optimizations. You can do this by specifying a size/speed trade-off level from 0 (speed) to 4 (size). This trade-off does not turn optimization phases on or off. Instead, its level is a weight factor that is used in the different optimization phases to influence the heuristics. The higher the level, the more the compiler focusses on code size optimization. To choose a trade-off value read the description below about which optimizations are affected and the impact of the different trade-off values.

Note that the trade-off settings are directions and there is no guarantee that these are followed. The compiler may decide to generate different code if it assessed that this would improve the result.

To specify the size/speed trade-off optimization level:

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

The Properties dialog appears.

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

In the right pane the Settings appear.

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

4. Select a trade-off level in the **Trade-off between speed and size** box.

See also [C compiler option `--tradeoff \(-t\)`](#)

Instruction Selection

Trade-off levels 0, 1 and 2: the compiler selects the instructions with the smallest number of cycles.

Trade-off levels 3 and 4: the compiler selects the instructions with the smallest number of bytes.

Switch Jump Chain versus Jump Table

Instruction selection for the `switch` statements follows different trade-off rules. A `switch` statement can result in a jump chain or a jump table. The compiler makes the decision between those by measuring and weighing bytes and cycles. This weigh is controlled with the trade-off values:

Trade-off value	Time	Size
0	100%	0%
1	75%	25%
2	50%	50%
3	25%	75%
4	0%	100%

Subscript Strength Reduction

Subscript strength reduction is turned off by default, because it is not possible for the PCP to automatically determine if it improves the generated code.

The total number of additional pointers of a particular type in a particular loop is limited to 4 for the PCP.

The performance increases when more subscript pointers can be allocated for an ideal situation. Ideal is when no registers are needed for other objects than subscripts. This is rarely the case, therefore the maximum number of registers is set to 4 GPRs.

Loop Optimization

For a top-loop, the loop is entered at the top of the loop. A bottom-loop is entered at the bottom. Every loop has a test and a jump at the bottom of the loop, otherwise it is not possible to create a loop. Some top-loops also have a conditional jump before the loop. This is only necessary when the number of loop iterations is unknown. The number of iterations might be zero, in this case the conditional jump jumps over the loop.

Bottom loops always have an unconditional jump to the loop test at the bottom of the loop.

Trade-off value	Try to rewrite top-loops to bottom-loops	Optimize loops for size/speed
0	no	speed
1	yes	speed
2	yes	speed
3	yes	size
4	yes	size

Example:

```
int a;

void i( int l, int m )
{
    int i;

    for ( i = m; i < l; i++ )
    {
        a++;
    }
    return;
}
```

Coded as a bottom loop (compiled with **--tradeoff=4**) is:

```
_3:      jg      _2          ;; unconditional jump to loop test at bottom
        ld.i    r5,0x1
```

```

    ldl.il  r7,@DPTR(_PCP_a)
    add.pi  r5,[_PCP_a]
    st.pi   r5,[_PCP_a]
    add.i   r3,0x1
_2:
    comp    r3,r1,cc_uc
    jg      _3,cc_slt

```

Coded as a top loop (compiled with **--tradeoff=0**) is:

```

    ldl.il  r7,@DPTR(_PCP_a)
    ld.pi   r5,[_PCP_a]
    comp    r3,r1,cc_uc ;; test for at least one loop iteration
    jg      _2,cc_sge   ;; can be omitted when number of iterations is known
_3:
    add.i   r5,0x1
    add.i   r3,0x1
    comp    r3,r1,cc_uc
    jg      _3,cc_slt
_2:

```

Automatic Function Inlining

You can enable automatic function inlining with the option **--optimize+=inline (-Oi)** or by using `#pragma optimize +inline`. This option is also part of the **-O3** predefined option set.

When automatic inlining is enabled, you can use the options **--inline-max-incr** and **--inline-max-size** (or their corresponding pragmas `inline_max_incr` / `inline_max_size`) to control automatic inlining. By default their values are set to -1. This means that the compiler will select a value depending upon the selected trade-off level. The defaults are:

Trade-off value	inline-max-incr	inline-max-size
0	999	50
1	50	25
2	20	20
3	10	10
4	0	0

For example with trade-off value 1, the compiler inlines all functions that are smaller or equal to 25 internal compiler units. After that the compiler tries to inline even more functions as long as the function will not grow more than 50%.

When these options/pragmas are set to a value ≥ 0 , the specified value is used instead of the values from the table above.

Static functions that are called only once, are always inlined, independent of the values chosen for `inline-max-incr` and `inline-max-size`.

Code Compaction

Trade-off levels 0 and 1: code compaction is disabled.

Trade-off level 2: only code compaction of matches outside loops.

Trade-off level 3: code compaction of matches outside loops, and matches inside loops of patterns that have an estimate execution frequency lower or equal to 10.

Trade-off level 4: code compaction of matches outside loops, and matches inside loops of patterns that have an estimate execution frequency lower or equal to 100.

For loops where the iteration count is unknown an iteration count of 10 is assumed.

For the execution frequency the compiler also accounts nested loops.

3.6.4. Static Stack Alignment Optimizations

The compiler aligns static stack sections so that they are not located over a PRAM page boundary. The linker locates the begin of the static stack at a PRAM page boundary to ensure that all aligned static stack sections are not located over a page boundary. This alignment restriction saves code by not having to reload the DPTR pointer when it already contains the correct page. The compiler can optimize DPTR updates for static stack accesses to static stack objects that are located in the same page.

The disadvantage is that data space is spilled for the alignment restriction. With the [C compiler option `--align-stack`](#) you can prevent or reduce the alignment gaps on the static stack. Of course less or no page pointer updates can then be optimized by the compiler which increasing the code size. This is a typical data size versus code size optimization.

By default all static stack sections are aligned on a power of 2 depending on its size. The static stack maximum alignment value must be a power of two in the range [1..64].

```
--align-stack=value
```

A value of 1 equals to no alignment optimizations. The default value is 64, which aligns all static stack sections. Also static stack sections that are larger than 64 get an alignment of 64.

3.7. Static Code Analysis

Static code analysis (SCA) is a relatively new feature in compilers. Various approaches and algorithms exist to perform SCA, each having specific pros and cons.

SCA Implementation Design Philosophy

SCA is implemented in the TASKING compiler based on the following design criteria:

- An SCA phase does not take up an excessive amount of execution time. Therefore, the SCA can be performed during a normal edit-compile-debug cycle.
- SCA is implemented in the compiler front-end. Therefore, no new makefiles or work procedures have to be developed to perform SCA.

- The number of emitted false positives is kept to a minimum. A false positive is a message that indicates that a correct code fragment contains a violation of a rule/recommendation. A number of warnings is issued in two variants, one variant when it is *guaranteed* that the rule is violated when the code is executed, and the other variant when the rules is *potentially* violated, as indicated by a preceding warning message.

For example see the following code fragment:

```
extern int some_condition(int);
void f(void)
{
    char buf[10];
    int i;

    for (i = 0; i <= 10; i++)
    {
        if (some_condition(i))
        {
            buf[i] = 0; /* subscript may be out of bounds */
        }
    }
}
```

As you can see in this example, if `i=10` the array `buf[]` might be accessed beyond its upper boundary, depending on the result of `some_condition(i)`. If the compiler cannot determine the result of this function at run-time, the compiler issues the warning "subscript is *possibly* out of bounds" preceding the CERT warning "ARR35: do not allow loops to iterate beyond the end of an array". If the compiler can determine the result, or if the `if` statement is omitted, the compiler can guarantee that the "subscript is out of bounds".

- The SCA implementation has real practical value in embedded system development. There are no real objective criteria to measure this claim. Therefore, the TASKING compilers support well known standards for safety critical software development such as the MISRA guidelines for creating software for safety critical automotive systems and secure "CERT C Secure Coding Standard" released by CERT. CERT is founded by the US government and studies internet and networked systems security vulnerabilities, and develops information to improve security.

Effect of optimization level on SCA results

The SCA implementation in the TASKING compilers has the following limitations:

- Some violations of rules will only be detected when a particular optimization is enabled, because they rely on the analysis done for that optimization, or on the transformations performed by that optimization. In particular, the constant propagation and the CSE/PRE optimizations are required for some checks. It is preferred that you enable these optimizations. These optimizations are enabled with the default setting of the optimization level (**-O2**).
- Some checks require cross-module inspections and violations will only be detected when multiple source files are compiled and linked together by the compiler in a single invocation.

3.7.1. C Code Checking: CERT C

The CERT C Secure Coding Standard provides rules and recommendations for secure coding in the C programming language. The goal of these rules and recommendations is to eliminate insecure coding practices and undefined behaviors that can lead to exploitable vulnerabilities. The application of the secure coding standard will lead to higher-quality systems that are robust and more resistant to attack.

For details about the standard, see the [CERT C Secure Coding Standard](#) web site. For general information about CERT secure coding, see www.cert.org/secure-coding.

Versions of the CERT C standard

Version 1.0 of the CERT C Secure Coding Standard is available as a book by Robert C. Seacord [Addison-Wesley]. Whereas the web site is a wiki and reflects the latest information, the book serves as a fixed point of reference for the development of compliant applications and source code analysis tools.

The rules and recommendations supported by the TASKING compiler reflect the version of the CERT web site as of June 1 2009.

The following rules/recommendations implemented by the TASKING compiler, are not part of the book: [PRE11-C](#), [FLP35-C](#), [FLP36-C](#), [MSC32-C](#)

For a complete overview of the supported CERT C recommendations/rules by the TASKING compiler, see [Chapter 14, CERT C Secure Coding Standard](#).

Priority and Levels of CERT C

Each CERT C rule and recommendation has an assigned *priority*. Three values are assigned for each rule on a scale of 1 to 3 for

- severity - how serious are the consequences of the rule being ignored
 1. low (denial-of-service attack, abnormal termination)
 2. medium (data integrity violation, unintentional information disclosure)
 3. high (run arbitrary code)
- likelihood - how likely is it that a flaw introduced by ignoring the rule could lead to an exploitable vulnerability
 1. unlikely
 2. probable
 3. likely
- remediation cost - how expensive is it to comply with the rule
 1. high (manual detection and correction)
 2. medium (automatic detection and manual correction)

3. low (automatic detection and correction)

The three values are then multiplied together for each rule. This product provides a measure that can be used in prioritizing the application of the rules. These products range from 1 to 27. Rules and recommendations with a priority in the range of 1-4 are level 3 rules (low severity, unlikely, expensive to repair flaws), 6-9 are level 2 (medium severity, probable, medium cost to repair flaws), and 12-27 are level 1 (high severity, likely, inexpensive to repair flaws).

The TASKING compiler checks most of the level 1 and some of the level 2 CERT C recommendations/rules.

For a complete overview of the supported CERT C recommendations/rules by the TASKING compiler, see [Chapter 14, CERT C Secure Coding Standard](#).

To apply CERT C code checking to your application

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

The Properties dialog appears.

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

In the right pane the Settings appear.

3. On the Tool Settings tab, select **C Compiler » CERT C Secure Coding**.

4. Make a selection from the **CERT C secure code checking** list.

5. If you selected **Custom**, expand the **Custom CERT C** entry and enable one or more individual recommendations/rules.

On the command line you can use the option `--cert`.

```
cpccp --cert={all | name [-name], ...}
```

With `--diag=cert` you can see a list of the available checks, or you can use a three-letter mnemonic to list only the checks in a particular category. For example, `--diag=pre` lists all supported checks in the preprocessor category.

3.7.2. C Code Checking: MISRA-C

The C programming language is a standard for high level language programming in embedded systems, yet it is considered somewhat unsuitable for programming safety-related applications. Through enhanced code checking and strict enforcement of best practice programming rules, TASKING MISRA-C code checking helps you to produce more robust code.

MISRA-C specifies a subset of the C programming language which is intended to be suitable for embedded automotive systems. It consists of a set of rules, defined in *MISRA-C:2004, Guidelines for the Use of the C Language in Critical Systems* (Motor Industry Research Association (MIRA), 2004).

The compiler also supports MISRA-C:1998, the first version of MISRA-C. You can select this version with the following C compiler option:

```
--misrac-version=1998
```

For a complete overview of all MISRA-C rules, see [Chapter 15, MISRA-C Rules](#).

Implementation issues

The MISRA-C implementation in the compiler supports nearly all rules. Only a few rules are not supported because they address documentation, run-time behavior, or other issues that cannot be checked by static source code inspection, or because they require an application-wide overview.

During compilation of the code, violations of the enabled MISRA-C rules are indicated with error messages and the build process is halted.

MISRA-C rules are divided in required rules and advisory rules. If rules are violated, errors are generated causing the compiler to stop. With the following options warnings, instead of errors, are generated for either or both the required rules and the advisory rules:

```
--misrac-required-warnings  
--misrac-advisory-warnings
```

Note that not all MISRA-C violations will be reported when other errors are detected in the input source. For instance, when there is a syntax error, all semantic checks will be skipped, including some of the MISRA-C checks. Also note that some checks cannot be performed when the optimizations are switched off.

Quality Assurance report

To ensure compliance to the MISRA-C rules throughout the entire project, the TASKING linker can generate 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. You can use this in your company's quality assurance system to provide proof that company rules for best practice programming have been applied in the particular project.

To apply MISRA-C code checking to your application

1. From the **Project** menu, select **Properties**
The Properties dialog appears.
2. In the left pane, expand **C/C++ Build** and select **Settings**.
In the right pane the Settings appear.
3. On the Tool Settings tab, select **C Compiler » MISRA-C**.
4. Select the **MISRA-C version** (2004 or 1998).
5. In the **MISRA-C checking** box select a MISRA-C configuration. Select a predefined configuration for conformance with the required rules in the MISRA-C guidelines.
6. (Optional) In the **Custom 2004** or **Custom 1998** entry, specify the individual rules.

On the command line you can use the [option --misrac](#).

```
cpcp --misrac={all | number [-number],...}
```

3.8. C Compiler Error Messages

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

F (Fatal errors)

After a fatal error the compiler immediately aborts compilation.

E (Errors)

Errors are reported, but the compiler continues compilation. No output files are produced unless you have set the [C compiler option --keep-output-files](#) (the resulting output file may be incomplete).

W (Warnings)

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

I (Information)

Information messages are always preceded by an error message. Information messages give extra information about the error.

S (System errors)

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

```
S9##: internal consistency check failed - please report
```

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

Display detailed information on diagnostics

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

The Problems view is added to the current perspective.

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

A popup menu appears.

3. Select **Detailed Diagnostics Info**.

A dialog box appears with additional information.

TASKING VX-toolset for PCP User Guide

On the command line you can use the [C compiler option `--diag`](#) to see an explanation of a diagnostic message:

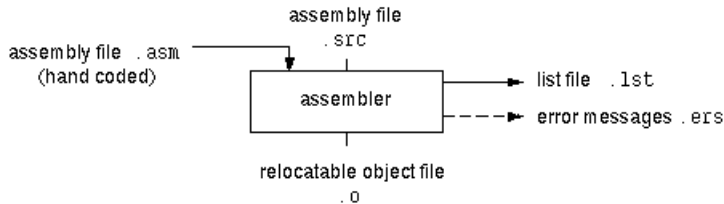
```
cpcp --diag=[format:]{all | number,...}
```

Chapter 4. Using the Assembler

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

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

The assembler takes the following files for input and output:



The following information is described:

- The assembly process.
- How to call the assembler and how to use its options. An extensive list of all options and their descriptions is included in [Section 8.2, *Assembler Options*](#).
- The various assembler optimizations.
- How to generate a list file.
- Types of assembler messages.

4.1. Assembly Process

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

Phases of the assembly process

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



The assembler integrates file inclusion and macro facilities. See [Section 2.10, *Macro Operations*](#) for more information.


4.2. Calling the Assembler

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

Building a project under Eclipse

You have several ways of building your project:

- Build Selected File(s) . This compiles and assembles the selected file(s) without calling the linker.
 1. In the C/C++ Projects view, select the files you want to compile.
 2. Right-click in the C/C++ Projects view and select **Build Selected File(s)**.
- Build Individual Project .

To build individual projects incrementally, select **Project » Build Project**.
- Rebuild Project . This builds every file in the project whether or not a file has been modified since the last build. A rebuild is a clean followed by a build.
 1. Select **Project » Clean...**
 2. Enable the option **Start a build immediately** and click **OK**.
- Build Automatically. This performs a build of all projects whenever any project file is saved, such as your makefile.

This way of building is not recommended for C/C++ development, but to enable this feature select **Project » Build Automatically** and ensure there is a check mark beside the **Build Automatically** menu item.

Select a target processor (core)

Processor options affect the invocation of all tools in the toolset. In Eclipse you only need to set them once. Based on the target processor, the compiler includes a special function register file. This is a regular include file which enables you to use virtual registers that are located in memory.

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

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

In the right pane the Processor page appears.
3. From the **Processor Selection** list, select a processor.

To access the assembler options

1. From the **Project** menu, select **Properties**
The Properties dialog appears.
2. In the left pane, expand **C/C++ Build** and select **Settings**.
In the right pane the Settings appear.
3. On the Tool Settings tab, select **Assembler**.
4. Select the sub-entries and set the options in the various pages.

Note that the options you enter in the Assembler page are not only used for hand-coded assembly files, but also for the assembly files generated by the compiler.

You can find a detailed description of all assembler options in [Section 8.2, Assembler Options](#).

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

```
aspcp [ [option]... [file]... ]...
```

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

4.3. How the Assembler Searches Include Files

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

1. If the `.INCLUDE` directive contains an absolute path name, the assembler looks for this file. If no path or a relative path is specified, the assembler looks in the same directory as the source file.
2. When the assembler did not find the include file, it looks in the directories that are specified in the **Assembler » Include Paths** page in the **C/C++ Build » Settings » Tool Settings** tab of the Project Properties dialog (equivalent to the `-I` command line option).
3. When the assembler did not find the include file (because it is not in the specified include directory or because no directory is specified), it looks in the path(s) specified in the environment variable `ASPCPINCL`.
4. When the assembler still did not find the include file, it finally tries the default include directory relative to the installation directory.

Example

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

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

You can call the assembler as follows:

TASKING VX-toolset for PCP User Guide

```
aspcp -Imyinclude test.asm
```

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

4.4. Assembler Optimizations

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

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

The Properties dialog appears.

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

In the right pane the Settings appear.

3. On the Tool Settings tab, select **Assembler » Optimization**.
4. Enable one or more optimizations.

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

Allow generic instructions (option **-Og/-OG**)

When this option is enabled, you can use generic instructions in your assembly source. The assembler tries to replace instructions by faster or smaller instructions. For example, the instruction `jc _label [,cc_X]` is replaced by a `jc, jl` or `jc.a` instruction.

By default this option is enabled. Because shorter instructions may influence the number of cycles, you may want to disable this option when you have written timed code. In that case the assembler encodes all instructions as they are.

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

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

4.5. Generating a List File

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

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

To generate a list file

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

The Properties dialog appears.

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

In the right pane the Settings appear.

3. On the Tool Settings tab, select **Assembler » List File**.
4. Enable the option **Generate list file**.
5. (Optional) Enable the options to include that information in the list file.

Example on the command line (Windows Command Prompt)

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

```
aspcp -l test.asm
```

See [Section 10.1, Assembler List File Format](#), for an explanation of the format of the list file.

4.6. Assembler Error Messages

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

F (Fatal errors)

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

E (Errors)

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

W (Warnings)

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

Display detailed information on diagnostics

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

The Problems view is added to the current perspective.

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

A popup menu appears.

3. Select **Detailed Diagnostics Info**.

TASKING VX-toolset for PCP User Guide

A dialog box appears with additional information.

On the command line you can use the [assembler option --diag](#) to see an explanation of a diagnostic message:

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

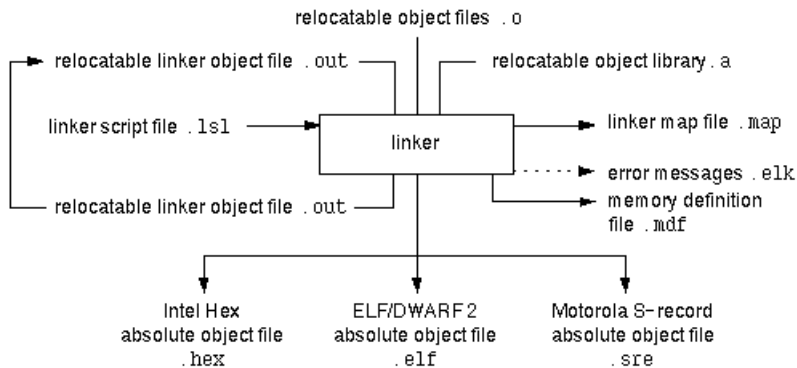
Chapter 5. Using the Linker

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

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

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

The linker takes the following files for input and output:



This chapter first describes the linking process. Then it describes how to call the linker and how to use its options. An extensive list of all options and their descriptions is included in [Section 8.3, Linker Options](#).

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

5.1. Linking Process

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

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

Terms used in the linking process

Term	Definition
Absolute object file	Object code in which addresses have fixed absolute values, ready to load into a target.
Address	A specification of a location in an address space.
Address space	The set of possible addresses. A core can support multiple spaces, for example in a Harvard architecture the addresses that identify the location of an instruction refer to <i>code</i> space, whereas addresses that identify the location of a data object refer to a <i>data</i> space.
Architecture	A description of the characteristics of a core that are of interest for the linker. This encompasses the address space(s) and the internal bus structure. Given this information the linker can convert logical addresses into physical addresses.
Copy table	<p>A section created by the linker. This section contains data that specifies how the startup code initializes the data and BSS sections. For each section the copy table contains the following fields:</p> <ul style="list-style-type: none"> • action: defines whether a section is copied or zeroed • destination: defines the section's address in RAM • source: defines the sections address in ROM, zero for BSS sections • length: defines the size of the section in MAUs of the destination space
Core	An instance of an architecture.
Derivative	The design of a processor. A description of one or more cores including internal memory and any number of buses.
Library	Collection of relocatable object files. Usually each object file in a library contains one symbol definition (for example, a function).
Logical address	An address as encoded in an instruction word, an address generated by a core (CPU).
LSL file	The set of linker script files that are passed to the linker.
MAU	Minimum Addressable Unit. For a given processor the number of bits between an address and the next address. This is not necessarily a byte or a word.
Object code	The binary machine language representation of the C source.
Physical address	An address generated by the memory system.
Processor	An instance of a derivative. Usually implemented as a (custom) chip, but can also be implemented in an FPGA, in which case the derivative can be designed by the developer.
Relocatable object file	Object code in which addresses are represented by symbols and thus relocatable.
Relocation	The process of assigning absolute addresses.

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

5.1.1. Phase 1: Linking

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

- *Header information:* Overall information about the file, such as the code size, name of the source file it was assembled from, and creation date.
- *Object code:* Binary code and data, divided into various named sections. Sections are contiguous chunks of code that have to be placed in specific parts of the memory. The program addresses start at zero for each section in the object file.
- *Symbols:* Some symbols are exported - defined within the file for use in other files. Other symbols are imported - used in the file but not defined (external symbols). Generally these symbols are names of routines or names of data objects.
- *Relocation information:* A list of places with symbolic references that the linker has to replace with actual addresses. When in the code an external symbol (a symbol defined in another file or in a library) is referenced, the assembler does not know the symbol's size and address. Instead, the assembler generates a call to a preliminary relocatable address (usually 0000), while stating the symbol name.
- *Debug information:* Other information about the object code that is used by a debugger. The assembler optionally generates this information and can consist of line numbers, C source code, local symbols and descriptions of data structures.

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

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

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

TASKING VX-toolset for PCP User Guide

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

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

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

5.1.2. Phase 2: Locating

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

Code modification

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

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

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

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

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

Output formats

The linker can produce its output in different file formats. The default ELF/DWARF 2 format (`.elf`) contains an image of the executable code and data, and can contain additional debug information. The Intel-Hex format (`.hex`) and Motorola S-record format (`.sre`) only contain an image of the executable code and data. You can specify a format with the options `--output (-o)` and `--chip-output (-c)`.

Controlling the linker

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

- The memory installed in the embedded target system:

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

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

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

- The underlying hardware architecture of the target processor.

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

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


See also [Section 5.7, Controlling the Linker with a Script](#).

5.2. Calling the Linker


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

Building a project under Eclipse

You have several ways of building your project:

- Build Individual Project .

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

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

1. Select **Project » Clean...**
2. Enable the option **Start a build immediately** and click **OK**.

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

TASKING VX-toolset for PCP User Guide

This way of building is not recommended, but to enable this feature select **Project » Build Automatically** and ensure there is a check mark beside the **Build Automatically** menu item.

To access the linker options

1. From the **Project** menu, select **Properties**
The Properties dialog appears.
2. In the left pane, expand **C/C++ Build** and select **Settings**.
In the right pane the Settings appear.
3. On the Tool Settings tab, select **Linker**.
4. Select the sub-entries and set the options in the various pages.

You can find a detailed description of all linker options in [Section 8.3, Linker Options](#).

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

```
lpcp [ [option]... [file]... ]...
```

When you are linking multiple files, either relocatable object files (.o) or libraries (.a), it is important to specify the files in the right order. This is explained in [Section 5.3, Linking with Libraries](#).

Example:

```
lpcp -dtc1165.lsl test.o
```

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

5.3. Linking with Libraries

There are two kinds of libraries: system libraries and user libraries.

System library

System libraries are stored in the directories:

```
<PCP installation path>\lib\pcp1 (PCP 1 libraries)  
<PCP installation path>\lib\pcp2 (PCP 2 libraries)
```

An overview of the system libraries is given in the following table:

Libraries	Description
libc[f].a	C libraries Optional letter: f = library compiled for __far memory

Libraries	Description
libfp[tf].a	Floating-point libraries Optional letter: t = trapping (control program option --fp-trap) f = library compiled for <code>__far</code> memory

To link the default C (system) libraries

1. From the **Project** menu, select **Properties**
The Properties dialog appears.
2. In the left pane, expand **C/C++ Build** and select **Settings**.
In the right pane the Settings appear.
3. On the Tool Settings tab, select **Linker » Libraries**.
4. Enable the option **Link default libraries**.
5. Enable or disable the option **Use trapped floating-point library**.

When you want to link system libraries from the command line, you must specify this with the option **--library (-l)**. For example, to specify the system library `libc.a`, type:

```
lpcp --library=c test.o
```

User library

You can create your own libraries. [Section 6.4, Archiver](#) describes how you can use the archiver to create your own library with object modules.

To link user libraries

1. From the **Project** menu, select **Properties**
The Properties dialog appears.
2. In the left pane, expand **C/C++ Build** and select **Settings**.
In the right pane the Settings appear.
3. On the Tool Settings tab, select **Linker » Libraries**.
4. Add your libraries to the **Libraries** box.

When you want to link user libraries from the command line, you must specify their filenames on the command line:

```
lpcp start.o mylib.a
```

If the library resides in a sub-directory, specify that directory with the library name:

TASKING VX-toolset for PCP User Guide

```
lpcp start.o mylibs\mylib.a
```

If you do not specify a directory, the linker searches the library in the current directory only.

Library order

The order in which libraries appear on the command line is important. By default the linker processes object files and libraries in the order in which they appear at the command line. Therefore, when you use a weak symbol construction, like `printf`, in an object file or your own library, you must position this object/library before the C library.

With the option **--first-library-first** you can tell the linker to scan the libraries from left to right, and extract symbols from the first library where the linker finds it. This can be useful when you want to use newer versions of a library routine:

```
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`.

5.3.1. How the Linker Searches Libraries

System libraries

You can specify the location of system libraries in several ways. The linker searches the specified locations in the following order:

1. The linker first looks in the **Library search path** that are specified in the **Linker » Libraries** page in the **C/C++ Build » Settings » Tool Settings** tab of the Project Properties dialog (equivalent to the **-L** command line option). If you specify the **-L** option without a pathname, the linker stops searching after this step.
2. When the linker did not find the library (because it is not in the specified library directory or because no directory is specified), it looks in the path(s) specified in the environment variables `LIBTC1V1_3` / `LIBTC1V1_3_1` / `LIBTC1V1_6`.
3. When the linker did not find the library, it tries the default `lib` directory relative to the installation directory (or a processor specific sub-directory).

User library

If you use your own library, the linker searches the library in the current directory only.

5.3.2. How the Linker Extracts Objects from Libraries

A library built with the TASKING archiver **arpcp** always contains an index part at the beginning of the library. The linker scans this index while searching for unresolved externals. However, to keep the index as small as possible, only the defined symbols of the library members are recorded in this area.

When the linker finds a symbol that matches an unresolved external, the corresponding object file is extracted from the library and is processed. After processing the object file, the remaining library index is searched. If after a complete search of the library unresolved externals are introduced, the library index will be scanned again. After all files and libraries are processed, and there are still unresolved externals and you did not specify the linker option `--no-rescan`, all libraries are rescanned again. This way you do not have to worry about the library order on the command line and the order of the object files in the libraries. However, this rescanning does not work for 'weak symbols'. If you use a weak symbol construction, like `printf`, in an object file or your own library, you must position this object/library before the C library.

The option `--verbose (-v)` shows how libraries have been searched and which objects have been extracted.

Resolving symbols

If you are linking from libraries, only the objects that contain symbols to which you refer, are extracted from the library. This implies that if you invoke the linker like:

```
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`.

It is possible to force a symbol as external (unresolved symbol) with the option `--extern (-e)`:

```
lpcp --extern=main mylib.a
```

In this case the linker searches for the symbol `main` in the library and (if found) extracts the object that contains `main`.

If this module contains new unresolved symbols, the linker looks again in `mylib.a`. This process repeats until no new unresolved symbols are found.

5.4. Incremental Linking

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

Incremental linking is only possible on the command line.

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

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

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

5.5. Importing Binary Files

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

For example:

```
#include <stdio.h>
__far extern char _lc_ub_my_mp3; /* linker labels */
__far extern char _lc_ue_my_mp3;
char* mp3 = &_lc_ub_my_mp3;

void main(void)
{
    int size = &_lc_ue_my_mp3 - &_lc_ub_my_mp3;
    int i;
    for (i=0;i<size;i++)
        putchar(mp3[i]);
}
```

Because the compiler does not know in which space the linker will locate the imported binary, you have to make sure the symbols refer to the same space in which the linker will place the imported binary. You do this by using the [memory qualifier](#) `__far`, otherwise the linker cannot bind your linker symbols.

Also note that if you want to use the export functionality of Eclipse, the binary file has to be part of your project.

5.6. Linker Optimizations

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

To enable or disable optimizations

1. From the **Project** menu, select **Properties**
The Properties dialog appears.
2. In the left pane, expand **C/C++ Build** and select **Settings**.
In the right pane the Settings appear.
3. On the Tool Settings tab, select **Linker » Optimization**.

4. Enable one or more optimizations.

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

Delete unreferenced sections (option -Oc/-OC)

This optimization removes unused sections from the resulting object file.

First fit decreasing (option -Ol/-OL)

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

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

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

Compress copy table (option -Ot/-OT)

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

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

Delete duplicate code (option -Ox/-OX)

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

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

5.7. Controlling the Linker with a Script

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

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

5.7.1. Purpose of the Linker Script Language

The Linker Script Language (LSL) serves three purposes:

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

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

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

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

The complete LSL syntax is described in Linker Script Language.

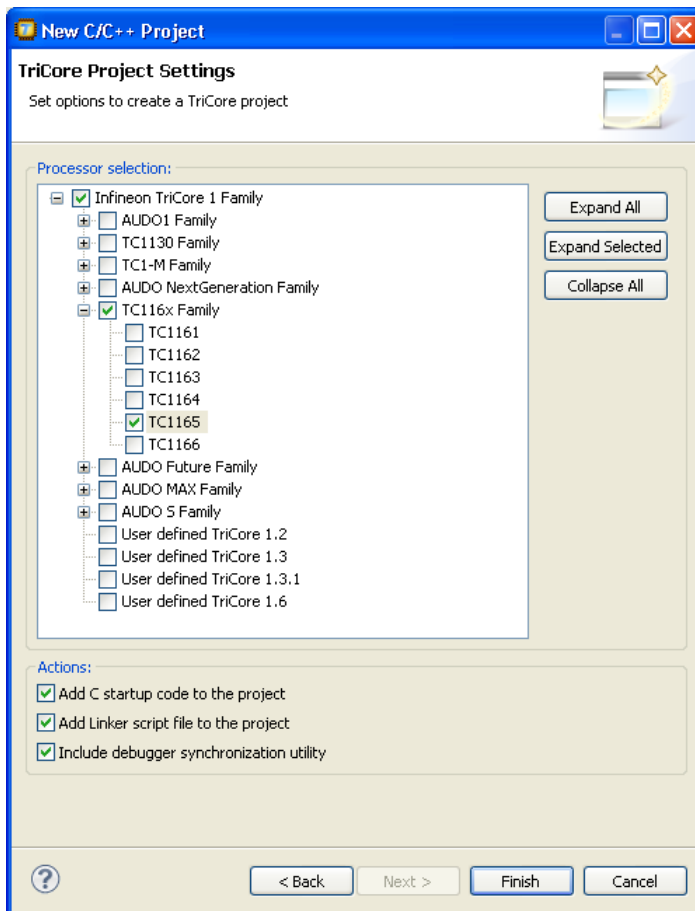
5.7.2. Eclipse and LSL

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

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

To add a generated Linker Script File to your project

1. From the **File** menu, select **File » New » TASKING VX-toolset for TriCore C/C++ Project**.
The New C/C++ Project wizard appears.
2. Fill in the project settings in each dialog and click **Next >** until the following dialog appears.



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

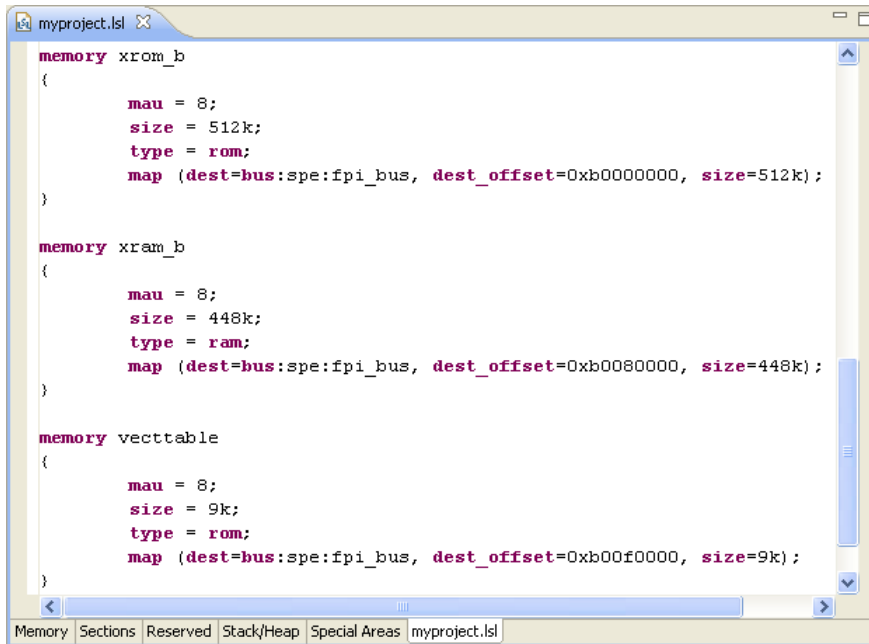
Eclipse creates your project and the file "project_name.lsl" in the project directory.

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

To change the Linker Script File in Eclipse


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

The project LSL file opens in the editor area with several tabs.



2. You can edit the LSL file directly in the `project_name.lsl` tab or make changes to the other tabs (Memory, Sections, ...).

*The LSL file is updated automatically according to the changes you make in the tabs. A * appears in front of the name of the LSL file to indicate that the file has changes.*

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

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

5.7.3. Structure of a Linker Script File

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

The architecture definition (required)

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

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

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

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

The derivative definition

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

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

Altium supplies LSL files for each derivative (*derivative.lsl*), 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.

The processor definition

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

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

The memory and bus definitions (optional)

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

The board specification

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

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

The section layout definition (optional)

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

Example: Skeleton of a Linker Script File

A linker script file that defines a derivative "X" based on the TC1V1.3 architecture, its external memory and how sections are located in memory, may have the following skeleton:

```
architecture TC1V1.3
{
    // Specification of the TC1V1.3 core architecture.
    // Written by Altium.
}

derivative X // derivative name is arbitrary
{
    // Specification of the derivative.
    // Written by Altium.
    core tc // always specify the core
    {
        architecture = TC1V1.3;
    }

    bus fpi_bus // internal bus
    {
        // maps to bus "fpi_bus" in "tc" core
    }

    // internal memory
}

processor spe // processor name is arbitrary
{
    derivative = X;

    // You can omit this part, except if you use a
    // multi-core system.
}

memory ext_name
{
    // external memory definition
}

section_layout spe:tc:linear // section layout
{
    // section placement statements
}
```

```

// sections are located in address space 'linear'
// of core 'tc' of processor 'spe'
}

```

Overview of LSL files delivered by Altium

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

LSL file	Description
<code>tc_arch.lsl</code>	Defines the base architecture (TC) for all cores. It includes the files <code>inttab.lsl</code> and <code>traptab.lsl</code> .
<code>inttab.lsl</code>	Defines the interrupt vector table. It is included in the file <code>tc_arch.lsl</code> .
<code>traptab.lsl</code>	Defines the trap vector table. It is included in the file <code>tc_arch.lsl</code> .
<code>tc1v1_3.lsl</code> <code>tc1v1_3_1.lsl</code> <code>tc1v1_6.lsl</code>	Extends the base architecture for cores TC1V1.3, TC1V1.3.1 and TC1V1.6. It includes the file <code>tc_arch.lsl</code> .
<code>derivative.lsl</code>	Defines the derivative and defines a single processor. Contains a memory definition and section layout. It includes one of the files <code>tcversion.lsl</code> or <code>pxbversion.lsl</code> . The selection of the derivative is based on your CPU selection (control program option <code>--cpu</code>).
<code>userdef13.lsl</code> <code>userdef131.lsl</code> <code>userdef16.lsl</code>	Defines a user defined derivative for cores TC1V1.3, TC1V1.3.1 or TC1V1.6 and defines a single processor.
<code>template.lsl</code>	This file is used by Eclipse as a template for the project LSL file. It includes the file <code>derivative.lsl</code> based on your CPU selection and contains a default specification of the external memory attached to the target processor.
<code>default.lsl</code>	Contains a default memory definition and section layout based on the <code>tc1165</code> derivative. This file is used on a command line invocation of the tools, when no CPU is selected (no option <code>--cpu</code>).
<code>extmem.lsl</code>	Template file with a specification of the external memory attached to the target processor.

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

5.7.4. The Architecture Definition

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

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

- space definitions: the logical address spaces and their properties

TASKING VX-toolset for PCP User Guide

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

Address spaces

A logical address space is a memory range for which the core has a separate way to encode an address into instructions. Most microcontrollers and DSPs support multiple address spaces. For example, the PCP has separate spaces for code and data. Normally, the size of an address space is 2^N , with N the number of bits used to encode the addresses.

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

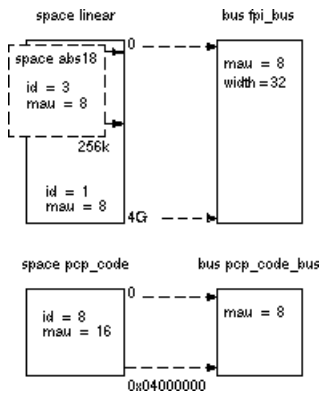
- one space is a subset of the other. These are often used for "small" absolute, and relative addressing.
- the addresses in the two address spaces represent different locations so they do not overlap. This means the core must have separate sets of address lines for the address spaces. For example, in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

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

Space	Id	MAU	Description	ELF sections
linear	1	1	Linear address space.	.text, .bss, .data, .rodata, table, istack, ustack
abs24	2	8	Absolute 24-bit addressable space	
abs18	3	8	Absolute 18-bit addressable space.	.zdata, .zbss
csa	4	8	Context Save Area	csa.*
pcp_code	8	16	PCP code	.pcptext
pcp_data	9	32	PCP data	.pcpdata

The TriCore architecture in LSL notation

The best way to program the architecture definition, is to start with a drawing. The following figure shows a part of the TriCore architecture:



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

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

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

space abs18
{
    id = 3;
    mau = 8;

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

space pcp_code
{
    id = 8;
    mau = 16;
    map (src_offset=0x00000000, dest_offset=0,
        size=0x04000000, dest=bus:pcp_code_bus);
}
```

TASKING VX-toolset for PCP User Guide

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

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

Next the two internal buses, named `fpi_bus` and `pcp_code_bus` must be defined in LSL:

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

bus pcp_code_bus
{
    mau = 8;
    width = 8;
}
```

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

```
architecture TC1V1.3
{
    // All code above goes here.
}
```

5.7.5. The Derivative Definition

Although you will probably not need to program the derivative definition (unless you are using multiple cores) it helps to understand the Linker Script Language and how the definitions are interrelated.

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

- core definition: an instance of a core architecture
- bus definition: the I/O buses of the core architecture
- memory definitions: internal (or on-chip) memory (in Eclipse this is called 'System memory')

Core

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

```

core tc
{
    architecture = TC1V1.3;
}

```

Bus

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

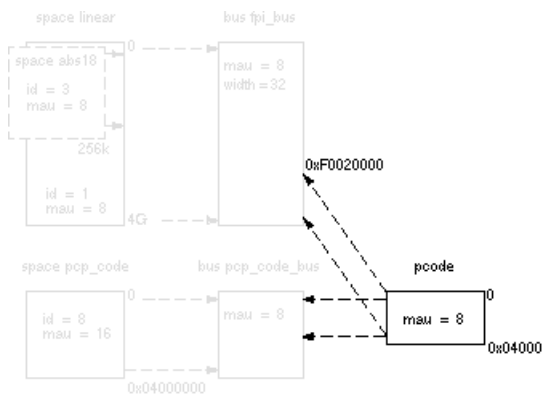
```

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

```

Memory

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



According to the drawing, the TriCore contains internal memory called `pcode` with a size `0x04000` (16 kB). This is physical memory which is mapped to the internal bus `pcp_code_bus` and to the `fpi_bus`, so both the `tc` unit and the PCP can access the memory:

```

memory pcode
{
    mau = 8;
    size = 16k;
    type = ram;
    map (dest=bus:tc:fpi_bus, dest_offset=0xF0020000,
        size=16k);
    map (dest=bus:tc:pcp_code_bus, size=16k);
}

```

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

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

5.7.6. The Processor Definition

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

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

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

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

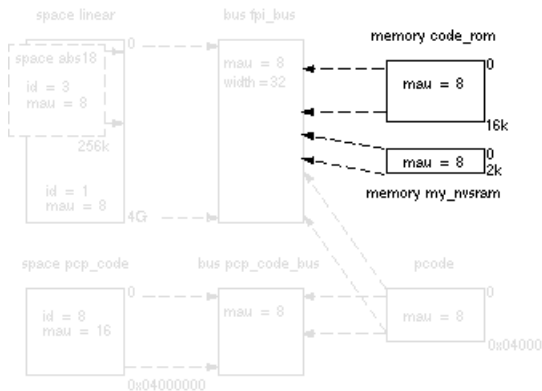
```
processor spe
{
    derivative = tc1165;
}
```

5.7.7. The Memory Definition

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

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

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

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

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

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

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

To add memory using Eclipse

1. Double-click on the file `project.lsl`.


The project LSL file opens in the editor area with several tabs.


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

A new line is added to the list of Memory.

3. Click in each field to change the type, name (for example `my_nvsram`) and sizes.

The LSL file is updated automatically according to the changes you make.

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

A  in front of a memory chip means that you cannot change this memory, because it is defined in a system LSL file.

5.7.8. The Section Layout Definition: Locating Sections

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

During compilation, the compiler divides the application into sections. Sections have a name, an indication (section type) in which address space it should be located and attributes like writable or read-only.

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

Example: section propagation through the toolset

To illustrate section placement, the following example of a C program (`bat.c`) is used. The program saves the number of times it has been executed in battery back-upped memory, and prints the number.

```
#define BATTERY_BACKUP_TAG 0xa5f0
#include <stdio.h>

int uninitialized_data;
int initialized_data = 1;
#pragma section data="non_volatile"
#pragma noclear
int battery_backup_tag;
int battery_backup_invok;
#pragma clear
#pragma endsection

void main (void)
{
    if (battery_backup_tag != BATTERY_BACKUP_TAG )
    {
        // battery back-upped memory area contains invalid data
        // initialize the memory
        battery_backup_tag = BATTERY_BACKUP_TAG;
        battery_backup_invok = 0;
    }
    printf( "This application has been invoked %d times\n",
           battery_backup_invok++);
}
```

The compiler assigns names and attributes to sections. With the `#pragma section data=non_volatile` the compiler's default section naming convention is overruled and a section with the name `.pcpdata.non_volatile` is defined. In this section the battery back-upped data is stored.

By default the compiler creates a section with the name ".pcpdata.data" of section type `data` to store uninitialized data objects. The attribute `clear` tells the linker that the section content should be filled with zeros at startup.

As a result of the `#pragma section data=non_volatile`, the data objects between the pragma pair are placed in a section with the name ".pcpdata.non_volatile". Note that uninitialized data sections are cleared at startup. However, battery back-upped sections should not be cleared and therefore we used `#pragma nclear`.

Section placement

The number of invocations of the example program should be saved in non-volatile (battery back-upped) memory. This is the memory `my_nvram` from the example in [Section 5.7.7, The Memory Definition](#).

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

```
section_layout ::pcp_data
{
    // Section placement statements
}
```

To locate sections, you must create a group in which you select sections from your program. For the battery back-up example, we need to define one group, which contains the section `.pcpdata.non_volatile`. All other sections are located using the defaults specified in the architecture definition. Section `.pcpdata.non_volatile` should be placed in non-volatile ram. To achieve this, the run address refers to our non-volatile memory called `my_nvram`.

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

Section placement from Eclipse


1. Double-click on the file `project.lsl`.

The project LSL file opens in the editor area with several tabs.

2. Open the **Sections** tab and click on the **Add...** button.

The Add New LSL Element dialog appears.

3. In the **New element** box, select **Section Layout** and click **Finish**.

A new section layout  appears. In the Section layout properties you can specify its characteristics. Note that you can add 'tags', which is just arbitrary text that can be added to a statement.

4. In the **Space** field of the Section layout properties, enter **pcp_data**.
5. Click on the `pcp_data` section layout and click on the **Add...** button.

TASKING VX-toolset for PCP User Guide

- In the **New element** box, select **Group** and in the **Parent** box select **section_layout ::pcp_data**. Click **Finish**.


An empty group element {} is added to the section layout. In the Group properties you can specify its characteristics.

- Click in the **Run address** field of the group and enter `mem:my_nvram`.
- In the Group properties part, select **Ordered**.
- Click the **Add...** button, select **Select Section(s)** and in the **Parent** box select the corresponding group. Click **Finish**.

A default select section element with the name "section_name" is added to the group. In the Section selection properties you can specify its characteristics.

- Click on the `section_name` and change it to `.pcpdata.non_valatile`.

The LSL file is updated automatically according to the changes you make.

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

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

For a complete description of the Linker Script Language, refer to [Chapter 11, Linker Script Language \(LSL\)](#).

5.8. Linker Labels

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

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

Label	Description
<code>_lc_ub_name</code> <code>_lc_b_name</code>	Begin of section <i>name</i> . Also used to mark the begin of the stack or heap or copy table.
<code>_lc_ue_name</code> <code>_lc_e_name</code>	End of section <i>name</i> . Also used to mark the end of the stack or heap.
<code>_lc_cb_name</code>	Start address of an overlay section in ROM.
<code>_lc_ce_name</code>	End address of an overlay section in ROM.
<code>_lc_gb_name</code>	Begin of group <i>name</i> . This label appears in the output file even if no reference to the label exists in the input file.

Label	Description
<code>_lc_ge_name</code>	End of group <i>name</i> . This label appears in the output file even if no reference to the label exists in the input file.
<code>_lc_s_name</code>	Variable <i>name</i> is mapped through memory in shared memory situations.

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

At C level, all linker labels start with `_lc_` (the PCP C compiler adds the label prefix `_PCP` and an extra underscore).

If you want to use linker labels in your C source for sections that have a dot (.) in the name, you have to replace all dots by underscores.

Additionally, the linker script file defines the following symbols:

Symbol	Description
<code>_lc_cp</code>	Start of copy table. Same as <code>_lc_ub_table</code> . The copy table gives the source and destination addresses of sections to be copied. This table will be generated by the linker only if this label is used.
<code>_lc_bh</code>	Begin of heap. Same as <code>_lc_ub_heap</code> .
<code>_lc_eh</code>	End of heap. Same as <code>_lc_ue_heap</code> .
<code>_PCP_lc_ub_heap_far</code>	Begin of PCP heap in TriCore address space linear. Same as <code>_lc_ub_pcp_heap_far</code> .
<code>_PCP_lc_ue_heap_far</code>	End of PCP heap in TriCore address space linear. Same as <code>_lc_ue_pcp_heap_far</code> .
<code>_PCP__lc_ub_heap</code>	Begin of PCP heap in address space <code>pcp_data</code> . Same as <code>_lc_ub_pcp_heap</code> .
<code>_PCP__lc_ue_heap</code>	End of PCP heap in address space <code>pcp_data</code> . Same as <code>_lc_ue_pcp_heap</code> .

Example: refer to a label with section name with dots from C

Suppose the C source file `foo.c` contains the following:

```
#pragma section myname
int myfunc(int a)
{
    /* some source lines */
    return 1;
}
#pragma endsection
```

This results in a section with the name `.pcptext.myname`.

In the following source file `main.c` all dots of the section name are replaced by underscores:

TASKING VX-toolset for PCP User Guide

```
#include <stdio.h>
extern char _lc_ub__pcptext_myname[];

void main(void)
{
    printf("The function myfunc is located at %x\n",
          &_lc_ub__pcptext_myname);
}
```

To prevent the linker error E106: unresolved external: `_PCP__lc_ub__pcptext_myname`, you must define this symbol in the LSL file as follows:

```
section_layout ::pcp_code
{
    "_PCP__lc_ub__pcptext_myname" := "_lc_ub__pcptext_myname";
}
```

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

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

Example: refer to a PCP variable from TriCore C source

When memory is shared between two or more cores, for instance TriCore and PCP, the addresses of variables (or functions) on that memory may be different for the cores. For the TriCore the variable will be defined and you can access it in the usual way. For the PCP, when you would use the variable directly in your TriCore source, this would use an incorrect address (PCP address). The linker can map the address of the variable from one space to another, if you prefix the variable name with `_lc_s_`.

When a symbol `foo` is defined in a PCP assembly source file, by default it gets the symbol name `foo`. To use this symbol from a TriCore C source file, write:

```
extern long _lc_s_foo;

void main(int argc, char **argv)
{
    _lc_s_foo = 7;
}
```

Example: refer to the heap

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. In the LSL file `tc_arch.lsl` a heap section is defined with the name `"pcp_heap"` (with the keyword `heap`). Symbol `_PCP__lc_ub_heap` is mapped to `_lc_ub_pcp_heap`. You can refer to the begin and end of the heap from your C source as follows:

```
#include <stdio.h>
extern char _lc_ub_heap[]; /* the compiler prefixes the label with _PCP_ */
extern char _lc_ue_heap[];
void main()
{
    printf( "Size of heap is %d\n",
           _lc_ue_heap - _lc_ub_heap );
}
```

In the C library the linker labels `_lc_ub_heap` and `_lc_ue_heap` are used in the 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.

From assembly you can refer to the end of the heap with:

```
.extern _PCP__lc_ue_heap    ; end of pcp_heap
```

5.9. Generating a Map File

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

To generate a map file

1. From the **Project** menu, select **Properties**
The Properties dialog appears.
2. In the left pane, expand **C/C++ Build** and select **Settings**.
In the right pane the Settings appear.
3. On the Tool Settings tab, select **Linker » Map File**.
4. Enable the option **Generate XML map file format (.mapxml) for map file viewer**.
5. (Optional) Enable the option **Generate map file (.map)**.
6. (Optional) Enable the options to include that information in the map file.

Example on the command line (Windows Command Prompt)

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

```
lpcp --map-file test.o
```

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

See [Section 10.2, Linker Map File Format](#), for an explanation of the format of the map file.

5.10. Linker Error Messages

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

F (Fatal errors)

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

E (Errors)

Errors are reported, but the linker continues linking and locating. No output files are produced unless you have set the [linker option--keep-output-files](#).

W (Warnings)

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

I (Information)

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

S (System errors)

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

```
S6##: message
```

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

Display detailed information on diagnostics

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

The Problems view is added to the current perspective.

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

A popup menu appears.

3. Select **Detailed Diagnostics Info**.

A dialog box appears with additional information.

On the command line you can use the [linker option --diag](#) to see an explanation of a diagnostic message:


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


Chapter 6. Using the Utilities

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

- ccpcp** A control program. The control program invokes all tools in the toolset and lets you quickly generate an absolute object file from C and/or assembly source input files. Eclipse uses the control program to call the compiler, assembler and linker.
- mkpcp** A utility program to maintain, update, and reconstruct groups of programs. The make utility looks whether files are out of date, rebuilds them and determines which other files as a consequence also need to be rebuilt.
- amk** The make utility which is used in Eclipse. It supports parallelism which utilizes the multiple cores found on modern host hardware.
- arpcp** An archiver. With this utility you create and maintain library files with relocatable object modules (.o) generated by the assembler.

6.1. Control Program

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

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

```
ccpcp [ option ]... [ file ]... ]...
```

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

Options

The control program accepts several command line options. If you specify an unknown option to the control program, the control program looks if it is an option for a specific tool. If so, it passes the option

TASKING VX-toolset for PCP User Guide

directly to the tool. However, it is recommended to use the control program options **--pass-*** (**-Wc**, **-Wa**, **-WI**) to pass arguments directly to tools.

For a complete list and description of all control program options, see [Section 8.4, Control Program Options](#).

Example with verbose output

```
ccpcp --verbose --cpu=tc1165 test.c
```

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

```
+ "path\ccpcp" -Ctc1165 --core=pcp2 -o cc3248a.src test.c
+ "path\aspcp" -Ctc1165 --core=pcp2 -o cc3248b.o cc3248a.src
+ "path\lpcp" -o test.elf -dtc1165.lsl -dextmem.lsl -D__CPU__=tc1165
  --map-file cc3248b.o "-Lpath\lib\pcp2" -lc -lfp
```

The control program produces unique filenames for intermediate steps in the compilation process (such as `cc3248a.src` and `cc3248b.o` in the example above) which are removed afterwards, unless you specify command line option **--keep-temporary-files (-t)**.

Example with argument passing to a tool

```
ccpcp --pass-compiler=-Oc test.c
```

The option **-Oc** is directly passed to the compiler.

6.2. Make Utility `mkpcp`

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

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

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

Make process

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

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

In addition, the make utility also reads the file `mkpcp.mk` which contains predefined rules and macros. See [Section 6.2.2, Writing a Makefile](#).

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

```
test.src : test.c                # dependency
          cpcp test.c            # rule

test.o   : test.src
          aspcp test.src

test.elf : test.o
          lpcp test.o -o test.elf --map-file -lc -lfp
```

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

Example

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

```
mkpcp test.elf

mkpcp -fmymake.mak test.elf
```

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

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

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

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

TASKING VX-toolset for PCP User Guide

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

If you compare this to the control program:

```
ccpcp test.c
```

This invocation has the same effect but now *all files* are recompiled (assembled, linked and located).

6.2.1. Calling the Make Utility

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

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

For example:

```
mkpcp test.elf
```

<i>target</i>	You can specify any target that is defined in the makefile. A target can also be one of the intermediate files specified in the makefile.
<i>macro=def</i>	Macro definition. This definition remains fixed for the mkpcp invocation. It overrides any regular definitions for the specified macro within the makefiles and from the environment. It is inherited by subordinate mkpcp 's but act as an environment variable for these. That is, depending on the -e setting, it may be overridden by a makefile definition.
<i>option</i>	For a complete list and description of all make utility options, see Section 8.5, Make Utility Options .

Exit status

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

6.2.2. Writing a Makefile

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

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

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

The makefile can contain a mixture of:

- [targets and dependencies](#)
- [rules](#)

- [macro definitions](#) or [functions](#)
- [conditional processing](#)
- [comment lines](#)
- [include lines](#)
- [export lines](#)

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

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

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

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

6.2.2.1. Targets and Dependencies

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

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

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

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

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

```
mkpcp  
mkpcp all  
mkpcp demo.elf final.elf
```

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

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

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

TASKING VX-toolset for PCP User Guide

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

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

Special targets

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

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

6.2.2.2. Makefile Rules

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

```
final.src : final.c # target and dependency
           move test.c final.c # rule1
           cpcp final.c # rule2
```

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

- @ does not echo the command line, except if **-n** is used.
- the make utility ignores the exit code of the command. Normally the make utility stops if a non-zero exit code is returned. This is the same as specifying the option **-i** on the command line or specifying the special `.IGNORE` target.

- + The make utility uses a shell or Windows command prompt (`cmd.exe`) to execute the command. If the '+' is not followed by a shell line, but the command is an MS-DOS command or if redirection is used (<, |, >), the shell line is passed to `cmd.exe` anyway.

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

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

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

Inline temporary files

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

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

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

Suffix targets

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

```
.SUFFIXES: .c
.c.o      :
          ccpcp -c $<
```

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

Implicit rules

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

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

TASKING VX-toolset for PCP User Guide

Example:

```
LIB =      -lc -lfp          # macro

prog.elf:  prog.o sub.o
          lpcp prog.o sub.o $(LIB) -o prog.elf

prog.o:    prog.c inc.h
          cpcp prog.c
          aspcp prog.src

sub.o:     sub.c inc.h
          cpcp sub.c
          aspcp sub.src
```

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

The following makefile uses implicit rules (from `mkpcp.mk`) to perform the same job.

```
LDFLAGS = -lc -lfp          # macro used by implicit rules
prog.elf: prog.o sub.o      # implicit rule used
prog.o:   prog.c inc.h      # implicit rule used
sub.o:    sub.c inc.h       # implicit rule used
```

6.2.2.3. Macro Definitions

A *macro* is a symbol name that is replaced with its definition before the makefile is executed. Although the macro name can consist of lower case or upper case characters, upper case is an accepted convention. The general form of a macro definition is:

```
MACRO = text
MACRO += and more text
```

Spaces around the equal sign are not significant. With the `+=` operator you can add a string to an existing macro. An extra space is inserted before the added string automatically.

To use a macro, you must access its contents:

```
$(MACRO)      # you can read this as
${MACRO}      # the contents of macro MACRO
```

If the macro name is a single character, the parentheses are optional. Note that the expansion is done recursively, so the body of a macro may contain other macros. These macros are expanded when the macro is actually used, not at the point of definition:

```
FOOD = $(EAT) and $(DRINK)
EAT = meat and/or vegetables
DRINK = water
export FOOD
```

The macro `FOOD` is expanded as `meat and/or vegetables and water` at the moment it is used in the export line, and the environment variable `FOOD` is set accordingly.

Predefined macros

Macro	Description
<code>MAKE</code>	Holds the value <code>mkpcp</code> . Any line which uses <code>MAKE</code> , temporarily overrides the option <code>-n</code> (Show commands without executing), just for the duration of the one line. This way you can test nested calls to <code>MAKE</code> with the option <code>-n</code> .
<code>MAKEFLAGS</code>	Holds the set of options provided to <code>mkpcp</code> (except for the options <code>-f</code> and <code>-d</code>). If this macro is exported to set the environment variable <code>MAKEFLAGS</code> , the set of options is processed before any command line options. You can pass this macro explicitly to nested <code>mkpcp</code> 's, but it is also available to these invocations as an environment variable.
<code>PRODDIR</code>	Holds the name of the directory where <code>mkpcp</code> is installed. You can use this macro to refer to files belonging to the product, for example a library source file. <code>DOPRINT = \$(PRODDIR)/lib/src/_doprint.c</code> When <code>mkpcp</code> is installed in the directory <code>c:/Tasking/bin</code> this line expands to: <code>DOPRINT = c:/Tasking/lib/src/_doprint.c</code>
<code>SHELLCMD</code>	Holds the default list of commands which are local to the <code>SHELL</code> . If a rule is an invocation of one of these commands, a <code>SHELL</code> is automatically spawned to handle it.
<code>\$</code>	This macro translates to a dollar sign. Thus you can use <code>\$\$</code> in the makefile to represent a single <code>"\$"</code> .

Dynamically maintained macros

There are several dynamically maintained macros that are useful as abbreviations within rules. It is best not to define them explicitly.

Macro	Description
<code>\$*</code>	The basename of the current target.
<code>\$<</code>	The name of the current dependency file.
<code>\$@</code>	The name of the current target.
<code>\$?</code>	The names of dependents which are younger than the target.
<code>\$!</code>	The names of all dependents.

The `$<` and `$*` macros are normally used for implicit rules. They may be unreliable when used within explicit target command lines. All macros may be suffixed with `F` to specify the Filename components (e.g. `${*F}`, `${@F}`). Likewise, the macros `$*`, `$<` and `$@` may be suffixed by `D` to specify the Directory component.

The result of the `$*` macro is always without double quotes (`"`), regardless of the original target having double quotes (`"`) around it or not.

The result of using the suffix **F** (Filename component) or **D** (Directory component) is also always without double quotes ("), regardless of the original contents having double quotes (") around it or not.

6.2.2.4. Makefile Functions

A function not only expands but also performs a certain operation. Functions syntactically look like macros but have embedded spaces in the macro name, e.g. '\$(match arg1 arg2 arg3)'. All functions are built-in and currently there are five of them: `match`, `separate`, `protect`, `exist` and `nexist`.

`match`

The `match` function yields all arguments which match a certain suffix:

```
$(match .o prog.o sub.o mylib.a)
```

yields:

```
prog.o sub.o
```

`separate`

The `separate` function concatenates its arguments using the first argument as the separator. If the first argument is enclosed in double quotes then '\n' is interpreted as a newline character, '\t' is interpreted as a tab, '\ooo' is interpreted as an octal value (where, `ooo` is one to three octal digits), and spaces are taken literally. For example:

```
$(separate "\n" prog.o sub.o)
```

results in:

```
prog.o
sub.o
```

Function arguments may be macros or functions themselves. So,

```
$(separate "\n" $(match .o $!))
```

yields all object files the current target depends on, separated by a newline string.

`protect`

The `protect` function adds one level of quoting. This function has one argument which can contain white space. If the argument contains any white space, single quotes, double quotes, or backslashes, it is enclosed in double quotes. In addition, any double quote or backslash is escaped with a backslash.

Example:

```
echo $(protect I'll show you the "protect" function)
```

yields:

```
echo "I'll show you the \"protect\" function"
```

exist

The `exist` function expands to its second argument if the first argument is an existing file or directory.

Example:

```
$(exist test.c ccpcp test.c)
```

When the file `test.c` exists, it yields:

```
ccpcp test.c
```

When the file `test.c` does not exist nothing is expanded.

nexist

The `nexist` function is the opposite of the `exist` function. It expands to its second argument if the first argument is not an existing file or directory.

Example:

```
$(nexist test.src ccpcp test.c)
```

6.2.2.5. Conditional Processing

Lines containing `ifdef`, `ifndef`, `else` or `endif` are used for conditional processing of the makefile. They are used in the following way:

```
ifdef macro-name
if-lines
else
else-lines
endif
```

The *if-lines* and *else-lines* may contain any number of lines or text of any kind, even other `ifdef`, `ifndef`, `else` and `endif` lines, or no lines at all. The `else` line may be omitted, along with the *else-lines* following it.

First the *macro-name* after the `ifdef` command is checked for definition. If the macro is defined then the *if-lines* are interpreted and the *else-lines* are discarded (if present). Otherwise the *if-lines* are discarded; and if there is an `else` line, the *else-lines* are interpreted; but if there is no `else` line, then no lines are interpreted.

When you use the `ifndef` line instead of `ifdef`, the macro is tested for not being defined. These conditional lines can be nested up to 6 levels deep.

You can also add tests based on strings. With `ifeq` the result is true if the two strings match, with `ifneq` the result is true if the two strings do not match. They are used in the following way:

```
ifeq(string1,string2)
if-lines
else
```

```
else-lines  
endif
```

6.2.2.6. Comment, Include and Export Lines

Comment lines

Anything after a "#" is considered as a comment, and is ignored. If the "#" is inside a quoted string, it is not treated as a comment. Completely blank lines are ignored.

```
test.src : test.c      # this is comment and is  
          ccpcp test.c # ignored by the make utility
```

Include lines

An *include line* is used to include the text of another makefile (like including a .h file in a C source). Macros in the name of the included file are expanded before the file is included. You can include several files. Include files may be nested.

```
include makefile2 makefile3
```

Export lines

An *export line* is used to export a macro definition to the environment of any command executed by the make utility.

```
GREETING = Hello  
export GREETING
```

This example creates the environment variable `GREETING` with the value `Hello`. The macro is exported at the moment the export line is read so the macro definition has to precede the export line.

6.3. Make Utility `amk`

amk is the make utility Eclipse uses to maintain, update, and reconstruct groups of programs. But you can also use it on the command line. Its features are a little different from **mkpcp**. The main difference compared to **mkpcp** and other make utilities, is that **amk** features parallelism which utilizes the multiple cores found on modern host hardware, hardening for path names with embedded white space and it has an (internal) interface to provide progress information for updating a progress bar. It does not use an external command shell (`/bin/sh`, `cmd.exe`) but executes commands directly.

The primary purpose of any make utility is to speed up the edit-build-test cycle. To avoid having to build everything from scratch even when only one source file changes, it is necessary to describe dependencies between source files and output files and the commands needed for updating the output files. This is done in a so called "makefile".

6.3.1. Makefile Rules

A makefile dependency rule is a single line of the form:

```
[target ...] : [prerequisite ...]
```

where *target* and *prerequisite* are path names to files. Example:

```
test.o : test.c
```

This states that target `test.o` depends on prerequisite `test.c`. So, whenever the latter is modified the first must be updated. Dependencies accumulate: prerequisites and targets can be mentioned in multiple dependency rules (circular dependencies are not allowed however). The command(s) for updating a target when any of its prerequisites have been modified must be specified with leading white space after any of the dependency rule(s) for the target in question. Example:

```
test.o :
    ccpcp test.c # leading white space
```

Command rules may contain dependencies too. Combining the above for example yields:

```
test.o : test.c
    ccpcp test.c
```

White space around the colon is not required. When a path name contains special characters such as `'`, `#` (start of comment), `=` (macro assignment) or any white space, then the path name must be enclosed in single or double quotes. Quoted strings can contain anything except the quote character itself and a newline. Two strings without white space in between are interpreted as one, so it is possible to embed single and double quotes themselves by switching the quote character.

When a target does not exist, its modification time is assumed to be very old. So, **amk** will try to make it. When a prerequisite does not exist possibly after having tried to make it, it is assumed to be very new. So, the update commands for the current target will be executed in that case. **amk** will only try to make targets which are specified on the command line. The default target is the first target in the makefile which does not start with a dot.

Static pattern rules

Static pattern rules are rules which specify multiple targets and construct the prerequisite names for each target based on the target name.

```
[target ...] : target-pattern : [prerequisite-patterns ...]
```

The *target* specifies the targets the rules applies to. The *target-pattern* and *prerequisite-patterns* specify how to compute the prerequisites of each target. Each target is matched against the *target-pattern* to extract a part of the target name, called the *stem*. This stem is substituted into each of the *prerequisite-patterns* to make the prerequisite names (one from each *prerequisite-pattern*).

Each pattern normally contains the character '%' just once. When the *target-pattern* matches a target, the '%' can match any part of the target name; this part is called the *stem*. The rest of the pattern must match exactly. For example, the target `foo.o` matches the pattern `%.o`, with `'foo'` as the stem. The targets `foo.c` and `foo.e1f` do not match that pattern.

The prerequisite names for each target are made by substituting the stem for the '%' in each prerequisite pattern.

Example:

```
objects = test.o filter.o
```

```
all: $(objects)
```

```
$(objects): %.o: %.c  
    ccpcp -c $< -o $@  
    echo the stem is $*
```

Here '\$<' is the automatic variable that holds the name of the prerequisite, '\$@' is the automatic variable that holds the name of the target and '\$*' is the stem that matches the pattern. Internally this translates to the following two rules:

```
test.o: test.c  
    ccpcp -c test.c -o test.o  
    echo the stem is test  
  
filter.o: filter.c  
    ccpcp -c filter.c -o filter.o  
    echo the stem is filter
```

Each target specified must match the target pattern; a warning is issued for each target that does not.

Special targets

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

Target	Description
<code>.DEFAULT</code>	If you call the make utility with a target that has no definition in the makefile, this target is built.

Target	Description
.DONE	When the make utility has finished building the specified targets, it continues with the rules following this target.
.INIT	The rules following this target are executed before any other targets are built.

6.3.2. Makefile Directives

Directives inside makefiles are executed while reading the makefile. When a line starts with the word "include" or "-include" then the remaining arguments on that line are considered filenames whose contents are to be inserted at the current line. "-include" will silently skip files which are not present. You can include several files. Include files may be nested.

Example:

```
include makefile2 makefile3
```

White spaces (tabs or spaces) in front of the directive are allowed.

6.3.3. Macro Definitions

A *macro* is a symbol name that is replaced with its definition before the makefile is executed. Although the macro name can consist of lower case or upper case characters, upper case is an accepted convention. When a line does not start with white space and contains the assignment operator '=', ':=' or '+=' then the line is interpreted as a macro definition. White space around the assignment operator and white space at the end of the line is discarded. Single character macro evaluation happens by prefixing the name with '\$'. To evaluate macros with names longer than one character put the name between parentheses '()' or curly braces '{}'. Macro names may contain anything, even white space or other macro evaluations.

Example:

```
DINNER = $(FOOD) and $(BEVERAGE)
FOOD = pizza
BEVERAGE = sparkling water
FOOD += with cheese
```

With the += operator you can add a string to an existing macro. An extra space is inserted before the added string automatically.

Macros are evaluated recursively. Whenever \$(DINNER) or \${DINNER} is mentioned after the above, it will be replaced by the text "pizza with cheese and sparkling water". The left hand side in a macro definition is evaluated before the definition takes place. Right hand side evaluation depends on the assignment operator:

- = Evaluate the macro at the moment it is used.
- := Evaluate the replacement text before defining the macro.

Subsequent '+=' assignments will inherit the evaluation behavior from the previous assignment. If there is none, then '+=' is the same as '='. The default value for any macro is taken from the environment. Macro definitions inside the makefile overrule environment variables. Macro definitions on the **amk** command line will be evaluated first and overrule definitions inside the makefile.

Macro	Description
\$	This macro translates to a dollar sign. Thus you can use "\$\$" in the makefile to represent a single "\$".
@	The name of the current target. When a rule has multiple targets, then it is the name of the target that caused the rule commands to be run.
*	The basename (or stem) of the current target. The stem is either provided via a static pattern rule or is calculated by removing all characters found after and including the last dot in the current target name. If the target name is 'test.c' then the stem is 'test' (if the target was not created via a static pattern rule).
<	The name of the first prerequisite.
MAKE	The amk path name (quoted if necessary). Optionally followed by the options -n and -s .
ORIGIN	The name of the directory where amk is installed (quoted if necessary).
SUBDIR	The argument of option -G . If you have nested makes with -G options, the paths are combined. This macro is defined in the environment (i.e. default macro value).

The @, * and < macros may be suffixed by 'D' to specify the directory component or by 'F' to specify the filename component. \$(@D) evaluates to the directory name holding the file\$(@F). \$(@D)/\$(@F) is equivalent to \$@. Note that on MS-Windows most programs accept forward slashes, even for UNC path names.

The result of the predefined macros @, * and < and 'D' and 'F' variants is not quoted, so it may be necessary to put quotes around it.

Note that stem calculation can cause unexpected values. For example:

\$@	\$*
/home/.wine/test	/home/
/home/test/.project	/home/test/
../file	/.

Macro string substitution

When the macro name in an evaluation is followed by a colon and equal sign as in

```
$(MACRO:string1=string2)
```

then **amk** will replace *string1* at the end of every word in \$(MACRO) by *string2* during evaluation. When \$(MACRO) contains quoted path names, the quote character must be mentioned in both the original string and the replacement string¹. For example:

```
$(MACRO:.o=".d")
```

6.3.4. Makefile Functions

A function not only expands but also performs a certain operation. The following functions are available:

¹Internally, **amk** tokenizes the evaluated text, but performs substitution on the original input text to preserve compatibility here with existing make implementations and POSIX.

\$(filter pattern ...,item ...)

The `filter` function filters a list of items using a pattern. It returns *items* that do match any of the *pattern* words, removing any items that do not match. The patterns are written using '%',

```
 ${filter %.c %.h, test.c test.h test.o readme.txt .project output.c}
```

results in:

```
 test.c test.h output.c
```

\$(filter-out pattern ...,item ...)

The `filter-out` function returns all *items* that do not match any of the *pattern* words, removing the items that do match one or more. This is the exact opposite of the `filter` function.

```
 ${filter-out %.c %.h, test.c test.h test.o readme.txt .project output.c}
```

results in:

```
 test.o readme.txt .project
```

\$(foreach var-name, item ..., action)

The `foreach` function runs through a list of items and performs the same *action* for each *item*. The *var-name* is the name of the macro which gets dynamically filled with an item while iterating through the *item* list. In the *action* you can refer to this macro. For example:

```
 ${foreach T, test filter output, ${T}.c ${T}.h}
```

results in:

```
 test.c test.h filter.c filter.h output.c output.h
```

6.3.5. Conditional Processing

Lines containing `ifdef`, `ifndef`, `else` or `endif` are used for conditional processing of the makefile. They are used in the following way:

```
ifdef macro-name
if-lines
else
else-lines
endif
```

The *if-lines* and *else-lines* may contain any number of lines or text of any kind, even other `ifdef`, `ifndef`, `else` and `endif` lines, or no lines at all. The `else` line may be omitted, along with the *else-lines* following it. White spaces (tabs or spaces) in front of preprocessing directives are allowed.

First the *macro-name* after the `ifdef` command is checked for definition. If the macro is defined then the *if-lines* are interpreted and the *else-lines* are discarded (if present). Otherwise the *if-lines* are discarded;

TASKING VX-toolset for PCP User Guide

and if there is an `else` line, the *else-lines* are interpreted; but if there is no `else` line, then no lines are interpreted.

When you use the `ifndef` line instead of `ifdef`, the macro is tested for not being defined. These conditional lines can be nested to any level.

You can also add tests based on strings. With `ifeq` the result is true if the two strings match, with `ifneq` the result is true if the two strings do not match. They are used in the following way:

```
ifeq(string1,string2)
if-lines
else
else-lines
endif
```

6.3.6. Makefile Parsing

amk reads and interprets a makefile in the following order:

1. When the last character on a line is a backslash (\) (i.e. without trailing white space) then that line and the next line will be concatenated, removing the backslash and newline.
2. The unquoted '#' character indicates start of comment and may be placed anywhere on a line. It will be removed in this phase.

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

3. Trailing white space is removed.
4. When a line starts with white space and it is not followed by a directive or preprocessing directive, then it is interpreted as a command for updating a target.
5. Otherwise, when a line contains the unquoted text '=', '+=' or ':=' operator, then it will be interpreted as a macro definition.
6. Otherwise, all macros on the line are evaluated before considering the next steps.
7. When the resulting line contains an unquoted ':' the line is interpreted as a dependency rule.
8. When the first token on the line is "include" or "-include" (which by now must start on the first column of the line), **amk** will execute the directive.
9. Otherwise, the line must be empty.

Macros in commands for updating a target are evaluated right before the actual execution takes place (or would take place when you use the **-n** option).

6.3.7. Makefile Command Processing

A line with leading white space (tabs or spaces) without a (preprocessing) directive is considered as a command for updating a target. When you use the option **-j** or **-J**, **amk** will execute the commands for

updating different targets in parallel. In that case standard input will not be available and standard output and error output will be merged and displayed on standard output only after the commands have finished for a target.

You can precede a command by one or more of the following characters:

- @ Do not show the command. By default, commands are shown prior to their output.
- Continue upon error. This means that **amk** ignores a non-zero exit code of the command.
- + Execute the command, even when you use option **-n** (dry run).
- | Execute the command on the foreground with standard input, standard output and error output available.

Built-in commands

Command	Description
<code>true</code>	This command does nothing. Arguments are ignored.
<code>false</code>	This command does nothing, except failing with exit code 1. Arguments are ignored.
<code>echo arg...</code>	Display a line of text.
<code>exit code</code>	Exit with defined code. Depending on the program arguments and/or the extra rule options '-' this will cause amk to exit with the provided code. Please note that 'exit 0' has currently no result.
<code>argfile file arg...</code>	Create an argument file suitable for the --option-file (-f) option of all the other tools. The first <code>argfile</code> argument is the name of the file to be created. Subsequent arguments specify the contents. An existing argument file is not modified unless necessary. So, the argument file itself can be used to create a dependency to options of the command for updating a target.

6.3.8. Calling the amk Make Utility

The invocation syntax of **amk** is:

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

For example:

```
amk test.elf
```

- target* You can specify any target that is defined in the makefile. A target can also be one of the intermediate files specified in the makefile.
- macro=def* Macro definition. This definition remains fixed for the **amk** invocation. It overrides any regular definitions for the specified macro within the makefiles and from the environment. It is not inherited by subordinate **amk**'s
- option* For a complete list and description of all **amk** make utility options, see [Section 8.6, Parallel Make Utility Options](#).

Exit status

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

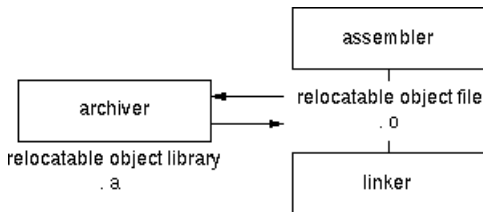
6.4. Archiver

The archiver **arpcp** is a program to build and maintain your own library files. A library file is a file with extension `.a` and contains one or more object files (`.o`) that may be used by the linker.

The archiver has five main functions:

- Deleting an object module from the library
- Moving an object module to another position in the library file
- Replacing an object module in the library or add a new object module
- Showing a table of contents of the library file
- Extracting an object module from the library

The archiver takes the following files for input and output:



The linker optionally includes object modules from a library if that module resolves an external symbol definition in one of the modules that are read before.

6.4.1. Calling the Archiver

You can create a library in Eclipse, which calls the archiver or you can call the archiver on the command line.


To create a library in Eclipse

Instead of creating a PCP absolute ELF file, you can choose to create a library. You do this when you create a new project with the New C/C++ Project wizard. (**File** ») select the option in the following dialog.

1. From the **File** menu, select **New** » **TASKING VX-toolset for PCP C Project**.

The New C/C++ Project wizard appears.

2. Enter a project name.
3. In the **Project type** box, select **TASKING PCP Library** and click **Next >**.

4. Follow the rest of the wizard and click **Finish**.
5. Add the files to your project.
6. Build the project as usual. For example, select **Project » Build Project** ()

Eclipse builds the library. Instead of calling the linker, Eclipse now calls the archiver.

Command line invocation

You can call the archiver from the command line. The invocation syntax is:

```
arpcp key_option [sub_option...] library [object_file]
```

<i>key_option</i>	With a key option you specify the main task which the archiver should perform. You must <i>always</i> specify a key option.
<i>sub_option</i>	Sub-options specify into more detail how the archiver should perform the task that is specified with the key option. It is not obligatory to specify sub-options.
<i>library</i>	The name of the library file on which the archiver performs the specified action. You must always specify a library name, except for the options -? and -V . When the library is not in the current directory, specify the complete path (either absolute or relative) to the library.
<i>object_file</i>	The name of an object file. You must always specify an object file name when you add, extract, replace or remove an object file from the library.

Options of the archiver utility

The following archiver options are available:

Description	Option	Sub-option
Main functions (key options)		
Replace or add an object module	-r	-a -b -c -u -v
Extract an object module from the library	-x	-v
Delete object module from library	-d	-v
Move object module to another position	-m	-a -b -v
Print a table of contents of the library	-t	-s0 -s1
Print object module to standard output	-p	
Sub-options		
Append or move new modules after existing module <i>name</i>	-a name	
Append or move new modules before existing module <i>name</i>	-b name	
Create library without notification if library does not exist	-c	
Preserve last-modified date from the library	-o	
Print symbols in library modules	-s{0 1}	
Replace only newer modules	-u	

Description	Option	Sub-option
Verbose	-v	
Miscellaneous		
Display options	-?	
Display version header	-V	
Read options from <i>file</i>	-f file	
Suppress warnings above level <i>n</i>	-wn	

For a complete list and description of all archiver options, see [Section 8.7, Archiver Options](#).

6.4.2. Archiver Examples

Create a new library

If you add modules to a library that does not yet exist, the library is created. To create a new library with the name `mylib.a` and add the object modules `cstart.o` and `calc.o` to it:

```
arpcp -r mylib.a cstart.o calc.o
```

Add a new module to an existing library

If you add a new module to an existing library, the module is added at the end of the module. (If the module already exists in the library, it is replaced.)

```
arpcp -r mylib.a mod3.o
```

Print a list of object modules in the library

To inspect the contents of the library:

```
arpcp -t mylib.a
```

The library has the following contents:

```
cstart.o
calc.o
mod3.o
```

Move an object module to another position

To move `mod3.o` to the beginning of the library, position it just before `cstart.o`:

```
arpcp -mb cstart.o mylib.a mod3.o
```

Delete an object module from the library

To delete the object module `cstart.o` from the library `mylib.a`:

```
arpcp -d mylib.a cstart.o
```


Extract all modules from the library

Extract all modules from the library `mylib.a`:

```
arpcp -x mylib.a
```


Chapter 7. Using the Debugger

This chapter describes the debugger and how you can run and debug a C or C++ application. This chapter only describes the TASKING specific parts.

7.1. Reading the Eclipse Documentation

Before you start with this chapter, it is recommended to read the Eclipse documentation first. It provides general information about the debugging process. This chapter guides you through a number of examples using the TASKING debugger with simulation as target.

You can find the Eclipse documentation as follows:

1. Start Eclipse.
2. From the **Help** menu, select **Help Contents**.
The help screen overlays the Eclipse Workbench.
3. In the left pane, select **C/C++ Development User Guide**.
4. Open the **Getting Started** entry and select **Debugging projects**.

This Eclipse tutorial provides an overview of the debugging process. Be aware that the Eclipse example does not use the TASKING tools and TASKING debugger.

7.2. Creating a Customized Debug Configuration

Before you can debug a project, you need a Debug launch configuration. Such a configuration, identified by a name, contains all information about the debug project: which debugger is used, which project is used, which binary debug file is used, which perspective is used, ... and so forth.

When you created your project, a default launch configuration for the TASKING simulator is available. If you used the Target Board Configuration wizard, also a default debug launch configuration for your target board is available. At any time you can change this configuration or create a custom debug configuration.

To debug or run a project, you need at least one opened and active project in your workbench. In this chapter, it is assumed that the `myproject` is opened and active in your workbench.

Customize your debug configuration

To change or create your own debug configuration follow the steps below.

1. From the **Run** menu, select **Debug Configurations...**
The Debug Configurations dialog appears.
2. In the left pane, select the configuration you want to change, for example, **TASKING Embedded C/C++ Application » myproject.simulator**.

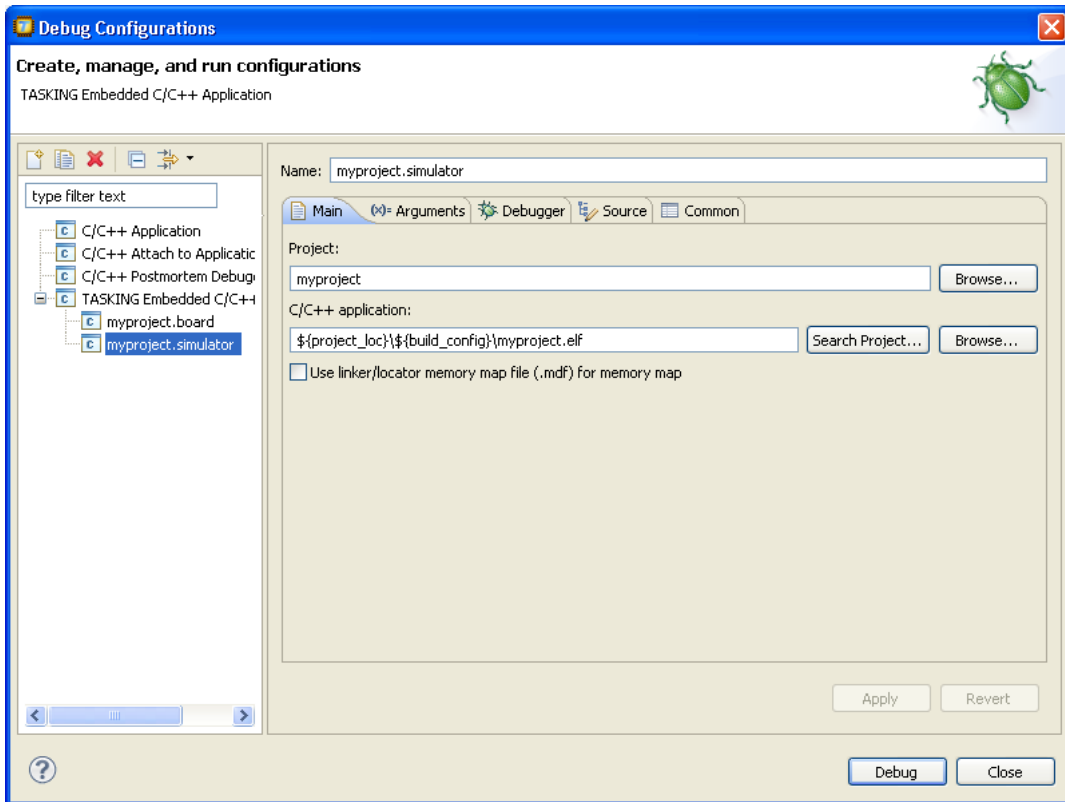
Or: click the **New launch configuration** button (📄) to add a new configuration.

The next dialog appears.

The dialog shows several tabs.

Main tab

On the **Main** tab, you can set the properties for the debug configuration such as a name for the configuration and the project and the application binary file which are used when you choose this configuration.



- **Name** is the name of the configuration. By default, this is the name of the project, optionally appended with `simulator` or `board`. You can give your configuration any name you want to distinguish it from the project name.
- In the **Project** field, you can choose the project for which you want to make a debug configuration. Because the project `myproject` is the active project, this project is filled in automatically. Click the `Browse...` button to select a different project. Only the *opened* projects in your workbench are listed.
- In the **C/C++ Application** field, you can choose the binary file to debug. The file `myproject.elf` is automatically selected from the active project.

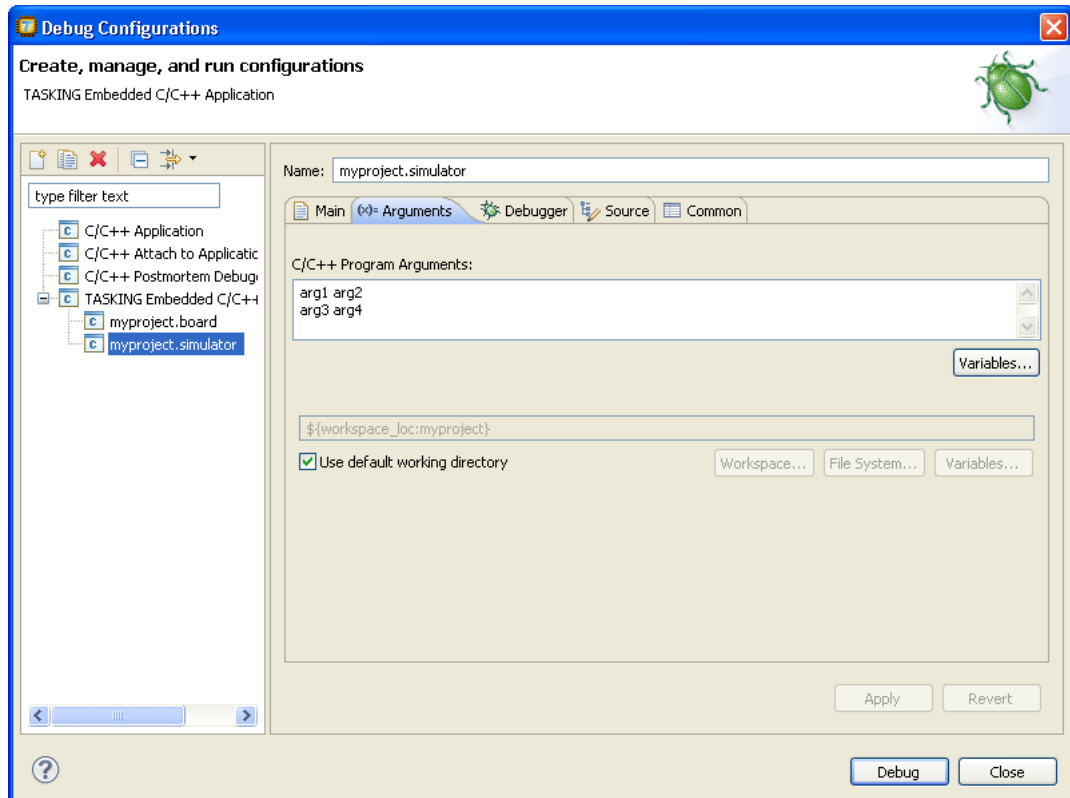
- You can use the option **Use linker/locator memory map file (.mdf) for memory map** to find errors in your application that cause access to non-existent memory or cause an attempt to write to read-only memory. When building your project, the linker/locator creates a memory description file (.mdf) file which describes the memory regions of the target you selected in your project properties. The debugger uses this file to initialize the debugging target.

This option is only useful in combination with a simulator as debug target. The debugger may fail to start if you use this option in combination with other debugging targets than a simulator.

Arguments tab

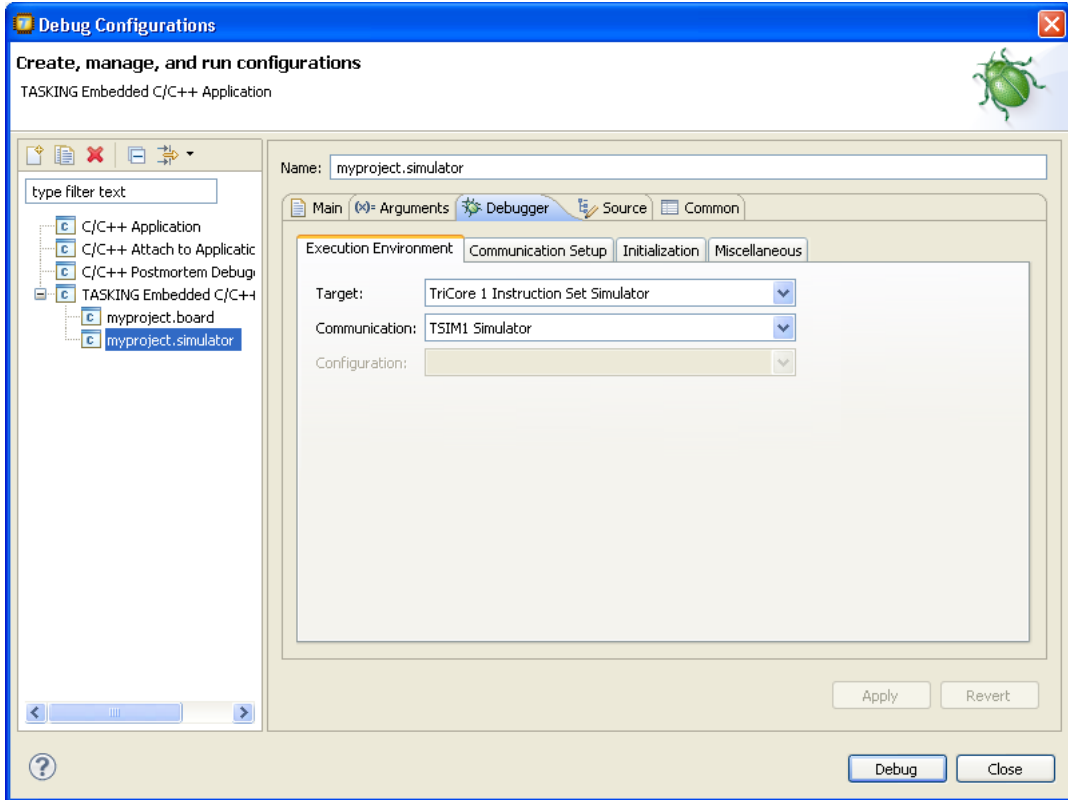
If your application's `main()` function takes arguments, you can pass them in this tab. Arguments are conventionally passed in the `argv[]` array. Because this array is allocated in target memory, make sure you have allocated sufficient memory space for it.

- In the C/C++ perspective double-click to open the file `cstart.c` in the **Startup Code Editor** view. At the bottom of the view select the **Configuration** tab, enable the option **Enable passing argc/argv to main()** and specify a **Buffer size for argv**.



Debugger tab

On the **Debugger** tab you can set the debugger options. You can choose which debugger should be used and with what options it should work. The Debugger tab itself contains several tabs.



- On the **Execution Environment** tab you can select on which target the application should be debugged. An application may run on an external evaluation board, or on a simulator using your own PC. For the evaluation board these settings should be the same as you specified in the Target Board Configuration wizard.
- On the **Communication Setup** tab you can select the type of communication (RS-232, TCP/IP, CAN) for execution environments. This tab is grayed out for the simulator.
- On the **Initialization** tab enable one or more of the following options:
 - **Initial download of program**

If enabled, the target application is downloaded onto the target. If disabled, only the debug information in the file is loaded, which may be useful when the application has already been downloaded (or flashed) earlier. If downloading fails, the debugger will shut down.
 - **Verify download of program**

If enabled, the debugger verifies whether the code and data has been downloaded successfully. This takes some extra time but may be useful if the connection to the target is unreliable.

- **Program flash when downloading**

If enabled, also flash devices are programmed (if necessary). Flash programming will not work when you use a simulator.

- **Reset target**

If enabled, the target is immediately reset after downloading has completed.

- **Goto main**

If enabled, only the C startup code is processed when the debugger is launched. The application stops executing when it reaches the first C instruction in the function `main()`. Usually you enable this option in combination with the option **Reset Target**.

- **Break on exit**

If enabled, the target halts automatically when the `exit()` function is called.

- **Reduce target state polling**

If you have set a breakpoint, the debugger checks the status of the target every *number* of seconds to find out if the breakpoint is hit. In this field you can change the polling frequency.

- On the **Miscellaneous** tab you can specify several file locations.

- **Debugger location**

The location of the debugger itself. This should not be changed.

- **FSS root directory**

The initial directory used by file system simulation (FSS) calls. See the description of the [FSS view](#).

- **ORTI file and KSM module**

If you wish to use the debugger's special facilities for OSEK kernels, specify the name of your ORTI file and that of your KSM module (shared library) in the appropriate edit boxes. See also the description of the [RTOS view](#).

- **GDI log file and Debug instrument log file**

You can use the options GDI log file and Debug instrument log file (if applicable) to control the generation of internal log files. These are primarily intended for use by or at the request of Altium support personnel.

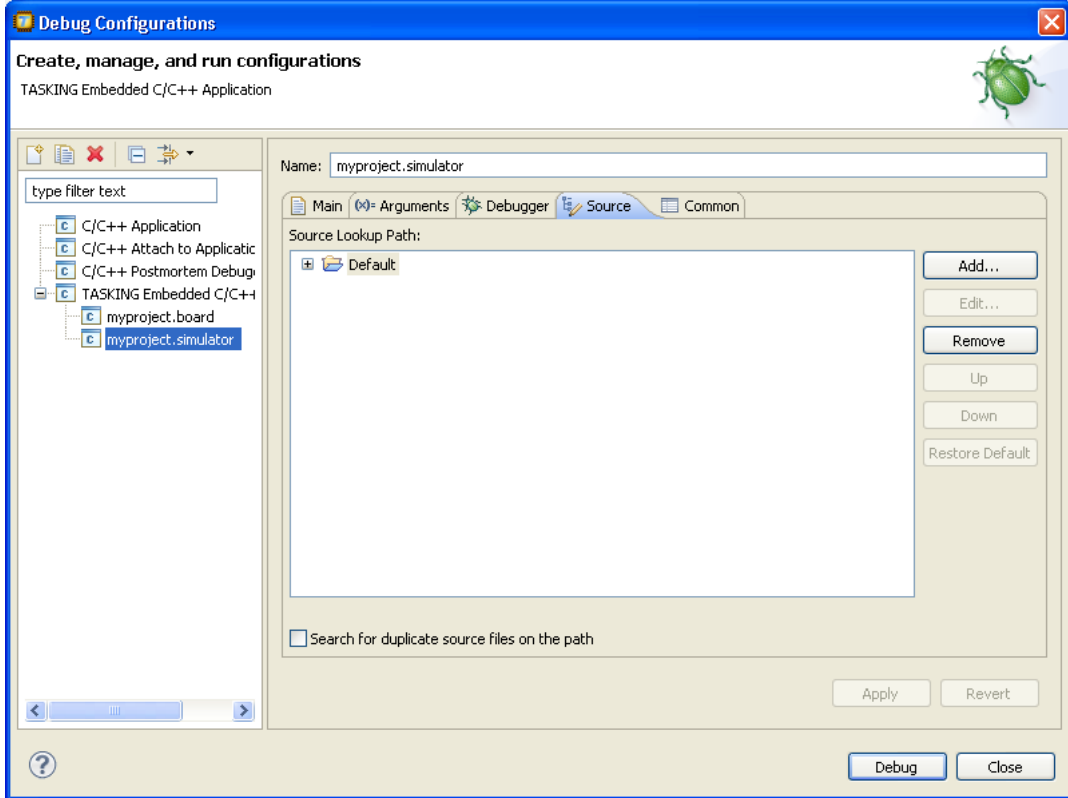
- **Cache target access**

Except when using a simulator, the debugger's performance is generally strongly dependent on the throughput and latency of the connection to the target. Depending on the situation, enabling this option may result in a noticeable improvement, as the debugger will then avoid re-reading registers

and memory while the target remains halted. However, be aware that this may cause the debugger to show the wrong data if tasks with a higher priority or external sources can influence the halted target's state.

Source tab

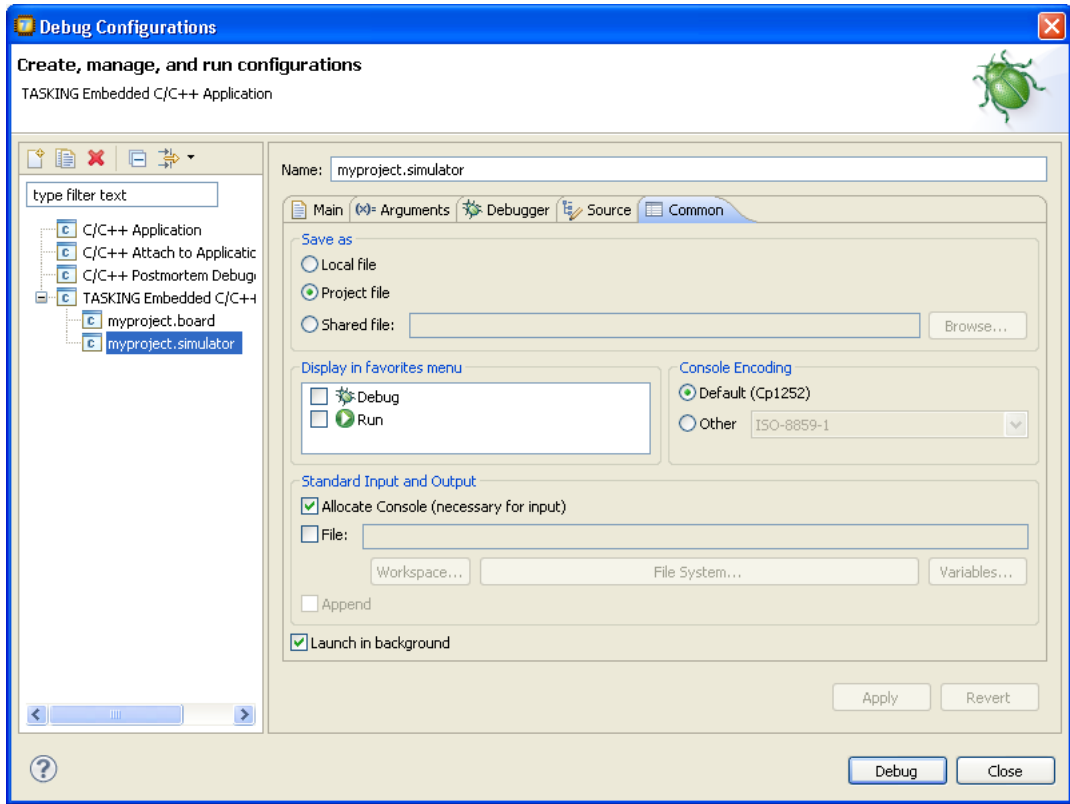
On the **Source** tab, you can add additional source code locations in which the debugger should search for debug data.



- Usually, the default source code location is correct.

Common tab

On the **Common** tab you can set additional launch configuration settings.



7.3. Troubleshooting

If the debugger does not launch properly, this is likely due to mistakes in the settings of the execution environment or to an improper connection between the host computer and the execution environment. Always read the notes for your particular execution environment.

Some common problems you may check for, are:

Problem	Solution
Wrong device name in the launch configuration	Make sure the specified device name is correct.
Invalid baud rate	Specify baud rate that matches the baud rate the execution environment is configured to expect.
No power to the execution environment.	Make sure the execution environment or attached probe is powered.
Wrong type of RS-232 cable.	Make sure you are using the correct type of RS-232 cable.
Cable connected to the wrong port on the execution environment or host.	Some target machines and hosts have several ports. Make sure you connect the cable to the correct port.

Problem	Solution
Conflict between communication ports.	A device driver or background application may use the same communications port on the host system as the debugger. Disable any service that uses the same port-number or choose a different port-number if possible.
Port already in use by another user.	The port may already be in use by another user on some UNIX hosts, or being allocated by a login process. Some target machines and hosts have several ports. Make sure you connect the cable to the correct port.

If the program state shown by the debugger appears to deviate from the true state, check that the linker option '[Include debugger synchronization utility](#)' is enabled.

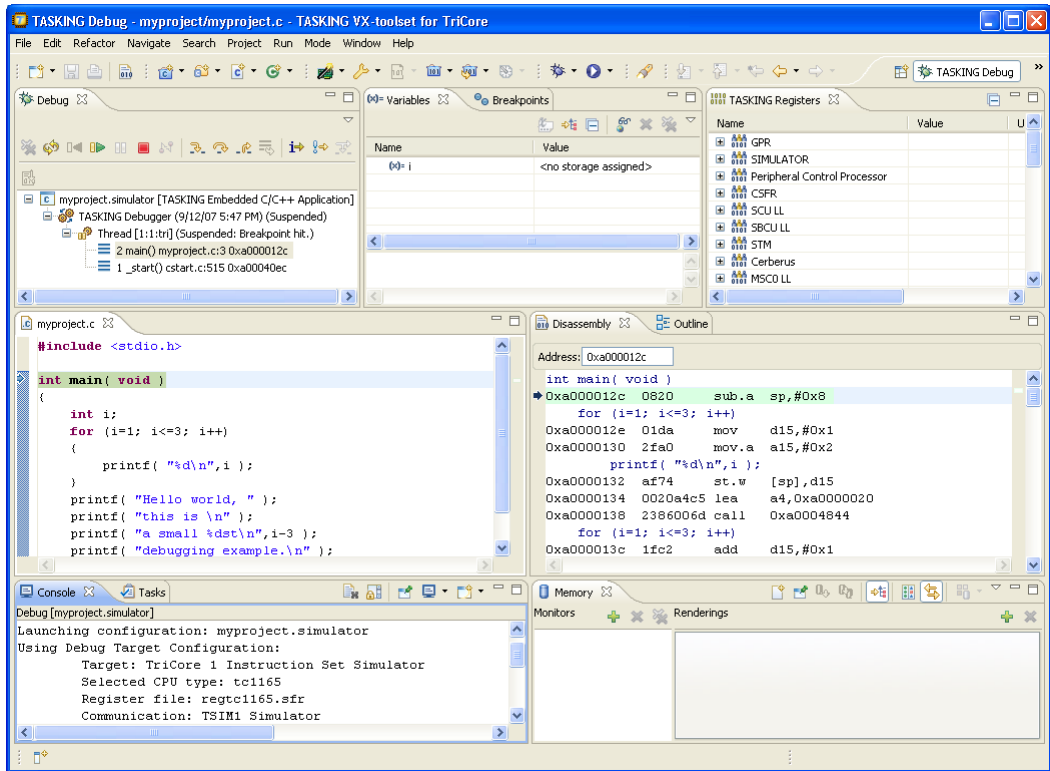
7.4. TASKING Debug Perspective

After you have launched the debugger, you are either asked if the TASKING Debug perspective should be opened or it is opened automatically. The Debug perspective consists of several views.

To open views in the Debug perspective:





1. Make sure the Debug perspective is opened
2. From the **Window** menu, select **Show View »**
3. Select a view from the menu or choose **Other...** for more views.

By default, the Debug perspective is opened with the following views:



7.4.1. Debug View

The Debug view shows the target information in a tree hierarchy shown below with a sample of the possible icons:

Icon	Session item	Description
	Launch instance	Launch configuration name and launch type
	Debugger instance	Debugger name and state
	Thread instance	Thread number and state
	Stack frame instance	Stack frame number, function, file name, and file line number

The number beside the thread label is a reference counter, not a thread identification number (TID).

Stack display













During debugging (running) the actual stack is displayed as it increases or decreases during program execution. By default, all views present information that is related to the current stack item (variables, memory, source code etc.). To obtain the information from other stack items, click on the item you want.

TASKING VX-toolset for PCP User Guide




The Debug view displays stack frames as child elements. It displays the reason for the suspension beside the thread, (such as end of stepping range, breakpoint hit, and signal received). When a program exits, the exit code is displayed.



The Debug view contains numerous functions for controlling the individual stepping of your programs and controlling the debug session. You can perform actions such as terminating the session and stopping the program. All functions are available from the right-click menu, though commonly used functions are also available from the toolbar in the Debug view.

Controlling debug sessions




Icon	Action	Description
	Remove all	Removes all terminated launches.
	Restart	Restarts the application. The target system is <i>not</i> reset.
	Reset target system	Resets the target system and restarts the application.
	Resume	Resumes the application after it was suspended (manually, breakpoint, signal).
	Suspend	Suspends the application (pause). Use the Resume button to continue.
	Relaunch	Right-click menu. Restarts the selected debug session when it was terminated. If the debug session is still running, a new debug session is launched.
	Reload current application	Reloads the current application without restarting the debug session. The application does restart of course.
	Terminate	Ends the selected debug session and/or process. Use Relaunch to restart this debug session, or start another debug session.
	Terminate all	Right-click menu. As terminate. Ends <i>all</i> debug sessions.
	Terminate and remove	Right-click menu. Ends the debug session and removes it from the Debug view.
	Terminate and Relaunch	Right-click menu. Ends the debug session and relaunches it. This is the same as choosing Terminate end then Relaunch.
	Disconnect	Detaches the debugger from the selected process (useful for debugging attached processes)

Stepping through the application

Icon	Action	Description
	Step into	Steps to the next source line or instruction
	Step over	Steps over a called function. The function is executed and the application suspends at the next instruction after the call.
	Step return	Executes the current function. The application suspends at the next instruction after the return of the function.

Icon	Action	Description
	Instruction stepping	Toggle. If enabled, the stepping functions are performed on instruction level instead of on C source line level.
	Interrupt aware stepping	Toggle. If enabled, the stepping functions do not step into an interrupt when it occurs.


Miscellaneous

Icon	Action	Description
	Copy Stack	Right-click menu. Copies the stack as text to the windows clipboard. You can paste the copied selection as text in, for example, a text editor.
	Edit <i>project...</i>	Right-click menu. Opens the debug configuration dialog to let you edit the current debug configuration.
	Edit Source Lookup...	Right-click menu. Opens the Edit Source Lookup Path window to let you edit the search path for locating source files.

7.4.2. Breakpoints View

You can add, disable and remove breakpoints by clicking in the marker bar (left margin) of the Editor view. This is explained in the Getting Started manual.

Description

The Breakpoints view shows a list of breakpoints that are currently set. The button bar in the Breakpoints view gives access to several common functions. The right-most button  opens the Breakpoints menu.

Types of breakpoints

To access the breakpoints dialog, add a breakpoint as follows:

1. Click the **Add TASKING Breakpoint** button (.

The Breakpoints dialog appears.

Each tab lets you set a breakpoint of a special type. You can set the following types of breakpoints:

- **File breakpoint**

The target halts when it reaches the specified line of the specified source file. Note that it is possible that a source line corresponds to multiple addresses, for example when a header file has been included into two different source files or when inlining has occurred. If so, the breakpoint will be associated with all those addresses.

- **Function**

The target halts when it reaches the first line of the specified function. If no source file has been specified and there are multiple functions with the given name, the target halts on all of those. Note that function breakpoints generally will not work on inlined instances of a function.

TASKING VX-toolset for PCP User Guide

- **Address**

The target halts when it reaches the specified instruction address.

- **Stack**

The target halts when it reaches the specified stack level.

- **Data**

The target halts when the given variable is read or written to, as specified.

- **Instruction**

The target halts when the given number of instructions has been executed.

- **Cycle**

The target halts when the given number of clock cycles has elapsed.

- **Timer**

The target halts when the given amount of time elapsed.

In addition to the type of the breakpoint, you can specify the condition that must be met to halt the program.

In the **Condition** field, type a condition. The condition is an expression which evaluates to 'true' (non-zero) or 'false' (zero). The program only halts on the breakpoint if the condition evaluates to 'true'.

In the **Ignore count** field, you can specify the number of times the breakpoint is ignored before the program halts. For example, if you want the program to halt only in the fifth iteration of a while-loop, type '4': the first four iterations are ignored.

7.4.3. File System Simulation (FSS) View

Description

The File System Simulation (FSS) view is automatically opened when the target requests FSS input or generates FSS output. The virtual terminal that the FSS view represents, follows the VT100 standard. If you right-click in the view area of the FSS view, a menu is presented which gives access to some self-explanatory functions.

VT100 characteristics

The `queens` example demonstrates some of the VT100 features. (You can find the `queens` example in the `<TriCore installation path>\examples` directory from where you can import it into your workspace.) Per debugging session, you can have more than one FSS view, each of which is associated with a positive integer. By default, the view "FSS #1" is associated with the standard streams `stdin`, `stdout`, `stderr` and `stdaux`. Other views can be accessed by opening a file named "terminal window <number>", as shown in the example below.

```
FILE * f3 = fopen("terminal window 3", "rw");
fprintf(f3, "Hello, window 3.\n");
fclose(f3);
```

You can set the initial working directory of the target application in the Debug configuration dialog (see also [Section 7.2, Creating a Customized Debug Configuration](#)):

1. On the **Debugger** tab, select the **Miscellaneous** sub-tab.
2. In the **FSS root directory** field, specify the FSS root directory.

The FSS implementation is designed to work without user intervention. Nevertheless, there are some aspects that you need to be aware of.

First, the interaction between the C library code (in the files `dbg*.c` and `dbg*.h`; see [Section 9.1.5, *dbg.h*](#)) and the debugger takes place via a breakpoint, which incidentally is not shown in the Breakpoints view. Depending on the situation this may be a hardware breakpoint, which may be in short supply.

Secondly, proper operation requires certain code in the C library to have debug information. This debug information should normally be present but might get lost when this information is stripped later in the development process.

7.4.4. Disassembly View

The Disassembly view shows target memory disassembled into instructions and / or data. If possible, the associated C / C++ source code is shown as well. The **Address** field shows the address of the current selected line of code.

To view the contents of a specific memory location, type the address in the **Address** field. If the address is invalid, the field turns red.

7.4.5. Expressions View

The Expressions view allows you to evaluate and watch regular C expressions.

To add an expression:

Click **OK** to add the expression.

1. Right-click in the Expressions View and select **Add Watch Expression**.

The Add Watch Expression dialog appears.

2. Enter an expression you want to watch during debugging, for example, the variable name "i"

If you have added one or more expressions to watch, the right-click menu provides options to **Remove** and **Edit** or **Enable** and **Disable** added expressions.

- You can access target registers directly using `#NAME`. For example `arr[#R0 << 3]` or `#TIMER3 = m++`. If a register is memory-mapped, you can also take its address, for example, `&#ADCIN`.
- Expressions may contain target function calls like for example `g1 + invert(&g2)`. Be aware that this will not work if the compiler has optimized the code in such a way that the original function code

does not actually exist anymore. This may be the case, for example, as a result of inlining. Also, be aware that the function and its callees use the same stack(s) as your application, which may cause problems if there is too little stack space. Finally, any breakpoints present affect the invoked code in the normal way.

7.4.6. Memory View

Use the Memory view to inspect and change process memory. The Memory view supports the same addressing as the C and C++ languages. You can address memory using expressions such as:

- `0x0847d3c`
- `(&y)+1024`
- `*ptr`

Monitors

To monitor process memory, you need to add a *monitor*.

1. In the Debug view, select a debug session. Selecting a thread or stack frame automatically selects the associated session.
2. Click the **Add Memory Monitor** button in the Memory Monitors pane.

The Monitor Memory dialog appears.

3. Type the address or expression that specifies the memory section you want to monitor and click **OK**.

The monitor appears in the monitor list and the Memory Renderings pane displays the contents of memory locations beginning at the specified address.

To remove a monitor:

1. In the Monitors pane, right-click on a monitor.
2. From the popup menu, select **Remove Memory Monitor**.

Renderings

You can inspect the memory in so-called *renderings*. A rendering specifies how the output is displayed: hexadecimal, ASCII, signed integer or unsigned integer. You can add or remove renderings per monitor. Though you cannot change a rendering, you can add or remove them:

1. Click the **Add Rendering** button in the Memory Renderings pane.

The Add Memory Rendering dialog appears.

2. Select the rendering you want (**Hex**, **ASCII**, **Signed Integer** or **Unsigned Integer**) and click **OK**.

To remove a rendering:

1. Right-click on a memory address in the rendering.

- From the popup menu, select **Remove Rendering**.

Changing memory contents

In a rendering you can change the memory contents. Simply type a new value.

Warning: Changing process memory can cause a program to crash.

The right-click popup menu gives some more options for changing the memory contents or to change the layout of the memory representation.

7.4.7. Compare Application View

You can use the Compare Application view to check if the downloaded application matches the application in memory. Differences may occur, for example, if you changed memory addresses in the Memory view.

- To check for differences, click the **Compare** button.

7.4.8. Heap View

With the Heap view you can inspect the status of the heap memory. This can be illustrated with the following example:

```
string = (char *) malloc(100);
strcpy ( string, "abcdefgh" );
free (string);
```

If you step through these lines during debugging, the Heap view shows the situation after each line has been executed. Before any of these lines has been executed, there is no memory allocated and the Heap view is empty.

- After the first line the Heap view shows that memory is occupied, the description tells where the block starts, how large it is (100 MAUs) and what its content is (0x0, 0x0, ...).
- After the second line, "abcdefgh" has been copied to the allocated block of memory. The description field of the Heap view again shows the actual contents of the memory block (0x61, 0x62, ...).
- The third line frees the memory. The Heap view is empty again because after this line no memory is allocated anymore.

7.4.9. Logging View

Use the Logging view to control the generation of internal log files. This view is intended mainly for use by or at the request of Altium support personnel.

7.4.10. RTOS View

The debugger has special support for debugging real-time operating systems (RTOSs). This support is implemented in an RTOS-specific shared library called a *kernel support module* (KSM) or *RTOS-aware debugging module* (RADM). Specifically, the TASKING VX-toolset for TriCore ships with a KSM supporting

the OSEK standard. You have to create your own OSEK Run Time Interface (ORTI) and specify this file on the **Miscellaneous** sub tab while configuring a customized debug configuration (see also [Section 7.2, Creating a Customized Debug Configuration](#)):

1. From the **Run** menu, select **Debug Configurations...**

The Debug Configurations dialog appears.

2. In the left pane, select the configuration you want to change, for example, **TASKING Embedded C/C++ Application » myproject.simulator**.


Or: click the **New launch configuration** button () to add a new configuration.

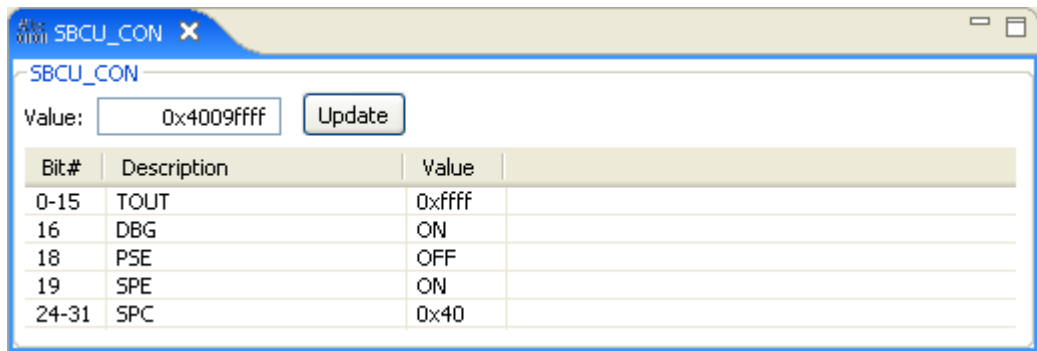
3. On the **Debugger** tab, select the **Miscellaneous** tab
4. In the **ORTI file** field, specify the name of your own ORTI file.

The debugger supports ORTI specifications v2.0 and v2.1.

7.4.11. TASKING Registers View

When first opened, the TASKING Registers view shows a number of *register groups*, which together contain all known registers. You can expand each group to see which registers they contain and examine the register's values while stepping through your application. This view has a number of features:

- While you step through the application, the registers involved in the step turn yellow.
- You can change each register's value.
- You can copy registers and/or groups to the windows clipboard: select the groups and/or individual registers, right-click on a register(group) and from the popup menu choose **Copy Registers**. You can paste the copied selection as text in, for example, a text editor.
- You can change the way the register value is displayed: right-click on a register(group) and from the popup menu choose the desired display mode (Natural, Hexadecimal, Decimal, Binary, Octal)
- For registers that are depicted with the icon , the menu entry **Symbolic Representation** is available in their right-click popup menu. This opens a new view which shows the internal fields of the register. (Alternatively, you can double-click on a register). For example, the SBCU_CON register from the Slow FPI Bus group may be shown as follows:



In this view you can set the individual values in the register, either by selecting a value from a drop-down box or by simply entering a value depending on the chosen field. To update the register with the new values, click the **Update** button.

- You can fully organize the register groups as you like: right-click on a register and from the popup menu use the menu items **Add Register Group...**, **Edit Register Group...** or **Remove Register Group**. This way you not only can choose which groups should be visible in the Register view, you can also create your own groups to which you add the registers of your interest.

To restore the original groups: right-click on a register and from the popup menu choose **Restore Register Groups**. Be aware: groups you have created will be removed, groups you have edited are restored to their original and groups you have deleted are placed back!

Viewing a register group in a separate view

For a better overview, you can open a register group in a separate view. To do so, double-click on the register group name. A new Register view is opened, showing all registers from the group. You can consider this view as a sub view of the Register view with roughly the same features.

7.4.12. Trace View

If tracing is enabled, the Trace view shows the code was most recently executed. For example, while you step through the application, the Trace view shows the executed code of each step. To enable tracing:

- From the **Run** menu, select **Trace**.

A check mark appears when tracing is enabled.

The view has three tabs, Source, Instruction and Raw, each of which represents the trace in a different way. However, not all target environments will support all three of these. The view is updated automatically each time the target halts.

7.5. PCP Simulator Configuration

To simulate the Peripheral Control Processor (PCP), the standard TriCore Instruction Set Simulator (ISS) starts a PCP plugin simulator (pcp). This is set up in a configuration file named `DConfig` that is located in the `etc` directory. This file tells the TriCore ISS to start up the PCP plugin with the specified options.

TASKING VX-toolset for PCP User Guide

The available command line options for the PCP plugin simulator are:

Option	Description	Default
-cmem_base <i>address</i>	Code memory address configuration	0xf0020000
-cmem_size <i>size</i>	Code memory size configuration	0x4000
-pram_base <i>address</i>	Parameter memory address configuration	0xf0010000
-pram_size <i>size</i>	Parameter memory size configuration	0x1000
-preg_base <i>address</i>	PCP register memory address configuration	0xf0003f00
-preg_size <i>size</i>	PCP register memory size configuration	0xcd
-psrn_base <i>address</i>	PCP service register node memory address configuration	0xf0003fd0
-psrn_size <i>size</i>	PCP service register node memory size configuration	0x30

If a non default derivative is used, it might be necessary to change the options in this file. The default derivative for the simulator is the tc1165.

Chapter 8. Tool Options

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

Tool options in Eclipse (Menu entry)

For each tool option that you can set from within Eclipse, a **Menu entry** description is available. In Eclipse you can customize the tools and tool options in the following dialog:

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

The Properties dialog appears.

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

In the right pane the Settings appear.

3. Open the **Tool Settings** tab.

You can set all tool options here.

Unless stated otherwise, all **Menu entry** descriptions expect that you have this Tool Settings tab open.

The following tables give an overview of all tool options on the Tool Settings tab in Eclipse with hyperlinks to the corresponding command line options (if available).

Global Options

Eclipse option	Description or option
Use global 'product directory' preference	Directory where the TASKING toolset is installed
Link the MIL representation of all modules	Control program option --mil-link / --mil-split
Treat warnings as errors	Control program option --warnings-as-errors
Keep temporary files	Control program option --keep-temporary-files (-t)
Verbose mode of control program	Control program option --verbose (-v)
Channel Configuration	
Prefix for global symbols	C compiler option --symbol-prefix
Allow channel to be interruptible	C compiler option --interrupt-enable
Preserve R7 flags 0..7	C compiler option --preserve-r7-flags

C Compiler

Eclipse option	Description or option
Preprocessing	
Automatic inclusion of '.sfr' file	C compiler option --no-tasking-sfr
Store preprocessor output in <file>.pre	Control program option --preprocess (-E) / --no-preprocessing-only
Keep comments in preprocessor output	Control program option --preprocess=+comments
Keep #line info in preprocessor output	Control program option --preprocess=-noline
Defined symbols	C compiler option --define
Pre-include files	C compiler option --include-file
Include Paths	
Include paths	C compiler option --include-directory
Language	
Comply to C standard	C compiler option --iso
Allow GNU C extensions	C compiler option --language=+gcc
Allow // comments in ISO C90 mode	C compiler option --language=+comments
Check assignment of string literal to non-const string pointer	C compiler option --language=-strings
Treat "char" variables as unsigned	C compiler option --uchar
Treat "int" bit-fields as signed	C compiler option --signed-bitfields
Allow optimization across volatile access	C compiler option --language=-volatile
Code Generation	
Maximum size for stack sections to align	C compiler option --align-stack
Generate channel entry table	C compiler option --no-channel-entry-table
Generate channel vectors	C compiler option --no-vector
Allocation	
Clear non-initialized global and static variables	C compiler option --no-clear
Optimization	
Optimization level	C compiler option --optimize
Trade-off between speed and size	C compiler option --tradeoff
Maximum size for code compaction	C compiler option --compact-max-size
Always inline function calls	C compiler option --inline
Maximum size increment when inlining (in %)	C compiler option --inline-max-incr
Maximum size for functions to always inline	C compiler option --inline-max-size
Custom Optimization	C compiler option --optimize

Eclipse option	Description or option
Debugging	
Generate symbolic debug information	C compiler option <code>--debug-info</code>
MISRA-C	
MISRA-C checking	C compiler option <code>--misrac</code>
MISRA-C version	C compiler option <code>--misrac-version</code>
Warnings instead of errors for required rules	C compiler option <code>--misrac-required-warnings</code>
Warnings instead of errors for advisory rules	C compiler option <code>--misrac-advisory-warnings</code>
Custom 1998 / Custom 2004	C compiler option <code>--misrac</code>
CERT C Secure Coding	
CERT C secure code checking	C compiler option <code>--cert</code>
Warnings instead of errors	C compiler option <code>--warnings-as-errors</code>
Custom CERT C	C compiler option <code>--cert</code>
Diagnostics	
Suppress C compiler warnings	C compiler option <code>--no-warnings=num</code>
Suppress all warnings	C compiler option <code>--no-warnings</code>
Perform global type checking on C code	C compiler option <code>--global-type-checking</code>
Miscellaneous	
Merge C source code with generated assembly	C compiler option <code>--source</code>
Additional options	C compiler options, Control program options

Assembler

Eclipse option	Description or option
Preprocessing	
Automatic inclusion of '.def' file	Assembler option <code>--no-tasking-sfr</code>
Defined symbols	Assembler option <code>--define</code>
Pre-include files	Assembler option <code>--include-file</code>
Include Paths	
Include paths	Assembler option <code>--include-directory</code>
Symbols	
Generate symbolic debug	Assembler option <code>--debug-info</code>
Case insensitive identifiers	Assembler option <code>--case-insensitive</code>
Emit local EQU symbols	Assembler option <code>--emit-locals+=equ</code>
Emit local non-EQU symbols	Assembler option <code>--emit-locals+=symbols</code>

Eclipse option	Description or option
Set default symbol scope to global	Assembler option --symbol-scope
Optimization	
Optimize generic instructions	Assembler option --optimize=+generics
Optimize instruction size	Assembler option --optimize=+instr-size
List File	
Generate list file	Control program option --list-files
List ...	Assembler option --list-format
List section summary	Assembler option --section-info=+list
Diagnostics	
Suppress warnings	Assembler option --no-warnings=num
Suppress all warnings	Assembler option --no-warnings
Display section summary	Assembler option --section-info=+console
Maximum number of emitted errors	Assembler option --error-limit
Miscellaneous	
Additional options	Assembler options

Linker

Eclipse option	Description or option
Libraries	
Use trapped floating-point library	Control program option --fp-trap
Link default libraries	Control program option --no-default-libraries
Rescan libraries to solve unresolved externals	Linker option --no-rescan
Libraries	The libraries are added as files on the command line.
Library search path	Linker option --library-directory
Data Objects	
Data objects	Linker option --import-object
Script File	
Defined symbols	Linker option --define
Linker script file (.lsl)	Linker option --lsl-file
Optimization	
Delete unreferenced sections	Linker option --optimize=c
Use a 'first-fit decreasing' algorithm	Linker option --optimize=l
Compress copy table	Linker option --optimize=t
Delete duplicate code	Linker option --optimize=x

Eclipse option	Description or option
Delete duplicate data	Linker option <code>--optimize=y</code>
Map File	
Generate map file (.map)	Control program option <code>--no-map-file</code>
Generate XML map file format (.mapxml) for map file viewer	Linker option <code>--map-file=file.mapxml:XML</code>
Include ...	Linker option <code>--map-file-format</code>
Diagnostics	
Suppress warnings	Linker option <code>--no-warnings=num</code>
Suppress all warnings	Linker option <code>--no-warnings</code>
Maximum number of emitted errors	Linker option <code>--error-limit</code>
Miscellaneous	
Strip symbolic debug information	Linker option <code>--strip-debug</code>
Link case insensitive	Linker option <code>--case-insensitive</code>
Do not use standard copy table for initialization	Linker option <code>--user-provided-initialization-code</code>
Include debugger synchronization utility	Linker option <code>--extern=_PCP_sync_on_halt</code>
Additional options	Linker options

8.1. C Compiler Options

This section lists all C compiler options.

Options in Eclipse versus options on the command line

Most command line options have an equivalent option in Eclipse but some options are only available on the command line. Eclipse invokes the compiler via the control program. Therefore, it uses the syntax of the control program to pass options and files to the C compiler. If there is no equivalent option in Eclipse, you can specify a command line option in Eclipse as follows:

- From the **Project** menu, select **Properties**
The Properties dialog appears.
- In the left pane, expand **C/C++ Build** and select **Settings**.
In the right pane the Settings appear.
- On the Tool Settings tab, select **C Compiler » Miscellaneous**.
- In the **Additional options** field, enter one or more command line options.

*Because Eclipse uses the control program, you have to precede the option with **-Wc** to pass the option via the control program directly to the C compiler.*

Be aware that some command line options are not useful in Eclipse or just do not have any effect. For example, the option **-n** sends output to stdout instead of a file and has no effect in Eclipse.

Short and long option names

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

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

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

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

C compiler option: `--align-stack`

Menu entry

1. Select **C Compiler » Code Generation**.
2. Enter a value in the **Maximum size for stack sections to align** field.

Command line syntax

`--align-stack=value`

Default: `--align-stack=64`

Description

Align static stack sections with 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 reloaded 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 align static stack sections with a size smaller than or equal to 32, enter:

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

The following invocation is not allowed, value is not a power of 2:

```
cpcp --align-stack=20 test.c // not allowed
```

Related information

-

C compiler option: `--cert`

Menu entry

1. Select **C Compiler » CERT C Secure Coding**.
2. Make a selection from the **CERT C secure code checking** list.
3. If you selected **Custom**, expand the **Custom CERT C** entry and enable one or more individual recommendations/rules.

Command line syntax

```
--cert={all | name[-name], ...}
```

Default format: all

Description

With this option you can enable one or more checks for CERT C Secure Coding Standard recommendations/rules. When you omit the argument, all checks are enabled. *name* is the name of a CERT recommendation/rule, consisting of three letters and two digits. Specify only the three-letter mnemonic to select a whole category. For the list of names you can use, see [Chapter 14, CERT C Secure Coding Standard](#).

On the command line you can use `--diag=cert` to see a list of the available checks, or you can use a three-letter mnemonic to list only the checks in a particular category. For example, `--diag=pre` lists all supported preprocessor checks.

Example

To enable the check for CERT rule STR30-C, enter:

```
cpcp --cert=str30 test.c
```

Related information

[Chapter 14, CERT C Secure Coding Standard](#)

[C compiler option `--diag`](#) (Explanation of diagnostic messages)

C compiler option: --check

Menu entry

-

Command line syntax

`--check`

Description

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

The compiler reports any warnings and/or errors.

This option is available on the command line only.

Related information

Assembler option `--check` (Check syntax)

C compiler option: `--compact-max-size`

Menu entry

1. Select **C Compiler » Optimization**.
2. In the **Maximum size for code compaction** field, enter the maximum size of a match.

Command line syntax

`--compact-max-size=value`

Default: 200

Description

This option is related to the compiler optimization `--optimize=+compact` (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 --optimize=+compact --compact-max-size=100 test.c
```

Related information

C compiler option `--optimize=+compact` (Optimization: code compaction)

C compiler option: `--core`

Menu entry

1. Select **C Compiler » Miscellaneous**.
2. Add the option `--core` to the **Additional options** field.

Command line syntax

`--core=core`

You can specify the following *core* arguments:

pcp1	PCP 1 syntax
pcp2	PCP 2 syntax

Default: derived from `--cpu`

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.

For more information see C compiler option `--cpu`.

Example

Specify a custom core:

```
cpcp --core=pcp2 test.c
```

Related information

[C compiler option `--cpu`](#) (Select processor)

C compiler option: `--cpu (-C)`

Menu entry

1. Expand **C/C++ Build** and select **Processor**.
2. From the **Processor Selection** list, select a processor or select **User defined TriCore**

Command line syntax

`--cpu=cpu`

`-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 this option the compiler always includes the special function register file `regcpu.sfr`, unless you disable the option **Automatic inclusion of '.sfr' file** on the **Preprocessing** page (option `--no-tasking-sfr`).

To avoid conflicts, make sure you specify the same target processor to the assembler (Eclipse and the control program do this automatically).

Example

To compile the file `test.c` for the TC1165 processor and use the SFR file `regtc1165.sfr`:

```
cpcp --cpu=tc1165 test.c
```

Related information

C compiler option `--core` (Select the PCP core)

C compiler option `--no-tasking-sfr` (Do not include SFR file)

Section 1.2.4, *Accessing Hardware from C*

C compiler option: `--debug-info (-g)`

Menu entry

1. Select **C Compiler » Debugging**.
2. To **generate symbolic debug information**, select **Default, Small set** or **Full**.
To disable the generation of debug information, select **None**.

Command line syntax

```
--debug-info[=suboption]
```

```
-g[suboption]
```

You can set the following suboptions:

small	1 / c	Emit small set of debug information.
default	2 / d	Emit default symbolic debug information.
all	3 / a	Emit full 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.

Small set of debug information

With this suboption only DWARF call frame information and type information are generated. This enables you to inspect parameters of nested functions. The type information improves debugging. You can perform a stack trace, but stepping is not possible because debug information on function bodies is not generated. You can use this suboption, for example, to compact libraries.

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 oversized assembler/object files.

Full debug information

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

Related information

-

C compiler option: --define (-D)

Menu entry

1. Select **C Compiler » Preprocessing**.

The Defined symbols box shows the symbols that are currently defined.

2. To define a new symbol, click on the **Add** button in the **Defined symbols** box.
3. Type the symbol definition (for example, demo=1)

Use the **Edit** and **Delete** button to change a macro definition or to remove a macro from the list.

Command line syntax

```
--define=macro_name[=macro_definition]
```

```
-Dmacro_name[=macro_definition]
```

Description

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

You can specify as many macros as you like. Simply use the **Add** button to add new macro definitions.

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

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

Example

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

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

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

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

Note that both invocations have the same effect.

The next example shows how to define a macro with arguments. Note that the macro name and definition are placed between double quotes because otherwise the spaces would indicate a new option.

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

Related information

[C compiler option **--undefine**](#) (Remove preprocessor macro)

[C compiler option **--option-file**](#) (Specify an option file)

C compiler option: `--dep-file`

Menu entry

Eclipse uses this option in the background to create a file with extension `.d` (one for every input file).

Command line syntax

```
--dep-file[=file]
```

Description

With this option you tell the compiler to generate dependency lines that can be used in a Makefile. In contrast to the option `--preprocess=+make`, the dependency information will be generated in addition to the normal output file.

By default, the information is written to a file with extension `.d` (one for every input file). When you specify a filename, all dependencies will be combined in the specified file.

Example

```
cpcp --dep-file=test.dep test.c
```

The compiler compiles the file `test.c`, which results in the output file `test.src`, and generates dependency lines in the file `test.dep`.

Related information

C compiler option `--preprocess=+make` (Generate dependencies for make)

C compiler option: `--diag`

Menu entry

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

The Problems view is added to the current perspective.

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

A popup menu appears.

3. Select **Detailed Diagnostics Info**.

A dialog box appears with additional information.

Command line syntax

```
--diag=[format:]{all | msg[-msg], ... }
```

You can set the following output formats:

html	HTML output.
rtf	Rich Text Format.
text	ASCII text.

Default format: text

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to stdout (normally your screen) and in the format you specify. The compiler does not compile any files. 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 (except for the CERT checks). If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas, or you can specify a range.

With `--diag=cert` you can see a list of the available CERT checks, or you can use a three-letter mnemonic to list only the checks in a particular category. For example, `--diag=pre` lists all supported preprocessor checks.

Example

To display an explanation of message number 282, enter:

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

This results in the following message and explanation:

TASKING VX-toolset for PCP User Guide

E282: unterminated comment

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

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

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

Related information

[Section 3.8, *C Compiler Error Messages*](#)

[C compiler option `--cert`](#) (Enable individual CERT checks)

C compiler option: `--error-file`

Menu entry

-

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

-

C compiler option: --global-type-checking

Menu entry

1. Select **C Compiler » Diagnostics**.
2. Enable the option **Perform global type checking on C code**.

Command line syntax

`--global-type-checking`

Description

The C compiler already performs type checking within each module. Use this option when you want the linker to perform type checking between modules.

Related information

-

C compiler option: --help (-?)

Menu entry

-

Command line syntax

```
--help[=item]
```

-?

You can specify the following arguments:

intrinsic	i	Show the list of intrinsic functions
options	o	Show extended option descriptions
pragmas	p	Show the list of supported pragmas
typedefs	t	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

-

C compiler option: --include-directory (-I)

Menu entry

1. Select **C Compiler » Include Paths**.

The Include paths box shows the directories that are added to the search path for include files.

2. To define a new directory for the search path, click on the **Add** button in the **Include paths** box.
3. Type or select a path.

Use the **Edit** and **Delete** button to change a path or to remove a path from the list.

Command line syntax

```
--include-directory=path,...
```

```
-Ipath,...
```

Description

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

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

1. The pathname in the C source file and the directory of the C source (only for #include files that are enclosed in "")
2. The path that is specified with this option.
3. The path that is specified in the environment variable `CPCPINC` when the product was installed.
4. The default directory `$(PRODDIR)\include` (unless you specified option `--no-stdinc`).

Example

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

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

You can call the compiler as follows:

```
cpcp --include-directory=myinclude test.c
```

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

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

Related information

C compiler option **--include-file** (Include file at the start of a compilation)

C compiler option **--no-stdinc** (Skip standard include files directory)

C compiler option: --include-file (-H)

Menu entry

1. Select **C Compiler » Preprocessing**.

The Pre-include files box shows the files that are currently included before the compilation starts.

2. To define a new file, click on the **Add** button in the **Pre-include files** box.
3. Type the full path and file name or select a file.

Use the **Edit** and **Delete** button to change a file name or to remove a file from the list.

Command line syntax

```
--include-file=file,...
```

```
-Hfile,...
```

Description

With this option you include one or more extra files 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 --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

C compiler option **--include-directory** (Add directory to include file search path)

C compiler option: `--inline`

Menu entry

1. Select **C Compiler » Optimization**.
2. Enable the option **Always inline function calls**.

Command line syntax

`--inline`

Description

With this option you instruct the compiler to inline calls to functions without the `__noinline` function qualifier whenever possible. This option has the same effect as a `#pragma inline` at the start of the source file.

This option can be useful to increase the possibilities for code compaction (C compiler option `--optimize=+compact`).

Example

To always inline function calls:

```
cpcp --optimize=+compact --inline test.c
```

Related information

C compiler option `--optimize=+compact` (Optimization: code compaction)

Section 1.8.3, *Inlining Functions: inline*

C compiler option: `--inline-max-incr` / `--inline-max-size`

Menu entry

1. Select **C Compiler » Optimization**.
2. In the **Maximum size increment when inlining** field, enter a value (default -1).
3. In the **Maximum size for functions to always inline** field, enter a value (default -1).

Command line syntax

```
--inline-max-incr=percentage (default: -1)  
--inline-max-size=threshold (default: -1)
```

Description

With these options you can control the automatic function inlining optimization process of the compiler. These options have only effect when you have enabled the inlining optimization (option `--optimize=+inline` or **Optimize most**).

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 -1, which means that the threshold depends on the option `--tradeoff`.

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. The default value is -1, which means that the value depends on the option `--tradeoff`.

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

C compiler option `--optimize=+inline` (Optimization: automatic function inlining)

Section 1.8.3, *Inlining Functions: inline*

Section 3.6.3, *Optimize for Size or Speed*

C compiler option: `--interrupt-enable`

Menu entry

1. Select **Global Options » Channel Configuration**.
2. Enable the option **Allow channel to be interruptible**.

Command line syntax

`--interrupt-enable`

Description

With this option the interrupt flag of a channel is enabled. The code generated is interruptible for channels that do not have common functions using static stack. The R7.IEN flag is enabled in the register context table of a channel. The IEN flag and CEN flag are set for each PRAM access or they are preserved when you specify [C compiler option `--preserve-r7-flags`](#).

Channels that have interrupts enabled must be linked separately. See [linker option `--link-only`](#) or [C compiler option `--mil`](#).

Related information

[Section 1.8.4.1, *Defining an Interrupt Service Routine: `__interrupt\(\)`*](#)

[Section 1.9.2, *Interruptible Code Generation*](#)

C compiler option: `--iso (-c)`

Menu entry

1. Select **C Compiler » Language**.
2. From the **Comply to C standard** list, select **ISO C99** or **ISO C90**.

Command line syntax

`--iso={90|99}`

`-c{90|99}`

Default: `--iso=99`

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 --iso=90 test.c
```

Related information

C compiler option `--language` (Language extensions)

C compiler option: `--keep-output-files (-k)`

Menu entry

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

Command line syntax

`--keep-output-files`

`-k`

Description

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

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

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

Example

```
cpcp --keep-output-files test.c
```

When an error occurs during compilation, the generated output file `test.src` will *not* be removed.

Related information

[C compiler option `--warnings-as-errors`](#) (Treat warnings as errors)

C compiler option: `--language (-A)`

Menu entry

1. Select **C Compiler » Language**.
2. Enable or disable one or more of the following options:
 - Allow GNU C extensions
 - Allow `//` comments in ISO C90 mode
 - Check assignment of string literal to non-const string pointer
 - Allow optimization across volatile access

Command line syntax

`--language=[flags]`

`-A[flags]`

You can set the following flags:

<code>+/-gcc</code>	<code>g/G</code>	enable a number of gcc extensions
<code>+/-comments</code>	<code>p/P</code>	<code>//</code> comments in ISO C90 mode
<code>+/-volatile</code>	<code>v/V</code>	don't optimize across volatile access
<code>+/-strings</code>	<code>x/X</code>	relaxed const check for string literals

Default: `-AGpVx`

Default (without flags): `-AGPVX`

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 `--language (-A)` without flags disables all language extensions.

GNU C extensions

The `--language=+gcc (-Ag)` option enables the following gcc language extensions:

- The identifier `__FUNCTION__` expands to the current function name.
- Alternative syntax for variadic macros.
- Alternative syntax for designated initializers.
- Allow zero sized arrays.

- Allow empty struct/union.
- Allow empty initializer list.
- Allow initialization of static objects by compound literals.
- The middle operand of a `? :` operator may be omitted.
- Allow a compound statement inside braces as expression.
- Allow arithmetic on void pointers and function pointers.
- Allow a range of values after a single case label.
- Additional preprocessor directive `#warning`.
- Allow comma operator, conditional operator and cast as lvalue.
- An inline function without `"static"` or `"extern"` will be global.
- An `"extern inline"` function will not be compiled on its own.
- An `__attribute__` directly following a struct/union definition relates to that tag instead of to the objects in the declaration.

For a more complete description of these extensions, you can refer to the UNIX gcc info pages (**info gcc**).

Comments in ISO C90 mode

With `--language=+comments (-Ap)` you tell the compiler to allow C++ style comments (`//`) in ISO C90 mode (option `--iso=90`). In ISO C99 mode this style of comments is always accepted.

Check assignment of string literal to non-const string pointer

With `--language=+strings (-Ax)` you disable warnings about discarded `const` qualifiers when a string literal is assigned to a non-const pointer.

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

Optimization across volatile access

With the `--language=+volatile (-Av)` option, the compiler will block optimizations when reading or writing a volatile object, by treating the access as a call to an unknown function. With this option you can prevent for example that code below the volatile object is optimized away to somewhere above the volatile object.

Example:

```
extern unsigned int variable;
extern volatile unsigned int access;

void TestFunc( unsigned int flag )
{
```

TASKING VX-toolset for PCP User Guide

```
    access = 0;
    variable |= flag;
    if( variable == 3 )
    {
        variable = 0;
    }
    variable |= 0x8000;
    access = 1;
}
```

Result with **--language=-volatile** (default):

```
_TestFunc    .proc    far
    movw     r11,_variable ; <== Moved across volatile access
    movw     _access,ZEROS ; <== Volatile access
    orw      r11,r2
    cmpw     r11,#0x3
    jmp      cc_ne,_2
    movw     r11,#0x0
_2:
    bset     r11.15
    movw     r12,#0x1
    movw     _access,r12 ; <== Volatile access
    movw     _variable,r11 ; <== Moved across volatile access
    ret
```

Result with **--language=+volatile**:

```
_TestFunc    .proc    far
    movw     _access,ZEROS ; <== Volatile access
    orw      _variable,r2
    movw     r11,#0x3
    cmpw     r11,_variable
    jmp      cc_ne,_2
    movw     _variable,ZEROS
_2:
    movw     r11,#0x8000
    orw      _variable,r11
    movw     r11,#0x1
    movw     _access,r11 ; <== Volatile access
    ret
```

Example

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

The compiler compiles in ISO C90 mode, accepts assignments of a constant string to a non-constant string pointer and does not allow C++ style comments.

Related information

C compiler option `--iso` (ISO C standard)

C compiler option: `--make-target`

Menu entry

-

Command line syntax

`--make-target=name`

Description

With this option you can overrule the default target name in the make dependencies generated by the options `--preprocess=+make` (`-Em`) and `--dep-file`. The default target name is the basename of the input file, with extension `.o`.

Example

```
cpcp --preprocess=+make --make-target=mytarget.o test.c
```

The compiler generates dependency lines with the default target name `mytarget.o` instead of `test.o`.

Related information

C compiler option `--preprocess=+make` (Generate dependencies for make)

C compiler option `--dep-file` (Generate dependencies in a file)

C compiler option: `--mil / --mil-split`

Menu entry

1. Select **C Compiler » Optimization**.
2. Enable the option **Build for application wide optimizations (MIL linking)**.

Command line syntax

```
--mil  
--mil-split[=file,...]
```

Description

With option `--mil` the C compiler skips the code generator phase and writes the optimized intermediate representation (MIL) to a file with the suffix `.mil`. The C compiler accepts `.mil` files as input files on the command line.

Option `--mil-split` does the same as option `--mil`, but in addition, the C compiler splits the MIL representation and writes it to separate files with suffix `.ms`. One file is written for each input file or MIL library specified on the command line. The `.ms` files are only updated on a change. The C compiler accepts `.ms` files as input files on the command line.

With option `--mil-split` you can perform application-wide optimizations during the frontend phase by specifying all modules at once, and still invoke the backend phase one module at a time to reduce the total compilation time.

Optionally, you can specify another filename for the `.ms` file the C compiler generates. Without an argument, the basename of the C source file is used to create the `.ms` filename. Note that if you specify a filename, you have to specify one filename for every input file.

Related information

[Section 3.1, *Compilation Process*](#)

Control program option `--mil-link / --mil-split`

C compiler option: `--misrac`

Menu entry

1. Select **C Compiler » MISRA-C**.
2. Make a selection from the **MISRA-C checking** list.
3. If you selected **Custom**, expand the **Custom 2004** or **Custom 1998** entry and enable one or more individual rules.

Command line syntax

`--misrac={all | nr[-nr]},...`

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

[Section 3.7.2, C Code Checking: MISRA-C](#)

[C compiler option `--misrac-advisory-warnings`](#)

[C compiler option `--misrac-required-warnings`](#)

[Linker option `--misrac-report`](#)

C compiler option: `--misrac-advisory-warnings` / `--misrac-required-warnings`

Menu entry

1. Select **C Compiler** » **MISRA-C**.
2. Make a selection from the **MISRA-C checking** list.
3. Enable one or both options **Warnings instead of errors for required rules** and **Warnings instead of errors for advisory rules**.

Command line syntax

`--misrac-advisory-warnings`

`--misrac-required-warnings`

Description

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

Related information

[Section 3.7.2, C Code Checking: MISRA-C](#)

[C compiler option `--misrac`](#)

[Linker option `--misrac-report`](#)

C compiler option: `--misrac-version`

Menu entry

1. Select **C Compiler » MISRA-C**.
2. Select the **MISRA-C version: 2004** or **1998**.

Command line syntax

```
--misrac-version={1998 | 2004}
```

Default: 2004

Description

MISRA-C rules exist in two versions: MISRA-C:1998 and MISRA-C:2004. By default, the C source is checked against the MISRA-C:2004 rules. With this option you can specify to check against the MISRA-C:1998 rules.

Related information

[Section 3.7.2, C Code Checking: MISRA-C](#)

C compiler option `--misrac`

C compiler option: `--no-channel-entry-table`

Menu entry

1. Select **C Compiler » Code Generation**.
2. Disable the option **Generate channel entry table**.

Command line syntax

`--no-channel-entry-table`

Description

When you use this option no channel start instruction is generated in the Channel Entry Table for an interrupt function. You can use this option when "Channel Start at Context PC" is used (CS.RCB=False) and no EXIT instruction resets the PC to the Channel Entry Table location of that interrupt channel (EP=0).

Related information

Section 1.8.4.1, *Defining an Interrupt Service Routine: `__interrupt()`*

C compiler option: `--no-clear`

Menu entry

1. Select **C Compiler » Allocation**.
2. Disable the option **Clear non-initialized global and static variables**.

Command line syntax

`--no-clear`

Description

Normally global/static variables are cleared at program startup. With option `--no-clear` 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`

C compiler option: `--no-partition`

Menu entry

1. Select **C Compiler » Optimization**.
2. In the **Optimization level** box, select **Custom Optimization**.
3. Select **C Compiler » Optimization » Custom Optimization**.
4. Disable the option **Automatic memory partitioning**.

Command line syntax

`--no-partition`

Description

With this option you tell the compiler to disable automatic memory partitioning.

Related information

[C compiler option `--optimize`](#) (Specify optimization level)

[Section 3.6.2, *Core Specific Optimizations \(backend\)*](#)

C compiler option: `--no-stdinc`

Menu entry

1. Select **C Compiler » Miscellaneous**.
2. Add the option `--no-stdinc` to the **Additional options** field.

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.

Related information

C compiler option `--include-directory` (Add directory to include file search path)

Section 3.4, *How the Compiler Searches Include Files*

C compiler option: `--no-tasking-sfr`

Menu entry

1. Select **C Compiler » Preprocessing**.
2. Disable the option **Automatic inclusion of '.sfr' file**.

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 selected on the **Processor** page (C compiler option `--cpu`).

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.

Related information

[C compiler option `--cpu` \(Select processor\)](#)

[Section 1.2.4, *Accessing Hardware from C*](#)

C compiler option: --no-vector

Menu entry

1. Select **C Compiler » Code Generation**.
2. Disable the option **Generate channel vectors**.

Command line syntax

`--no-vector`

Description

With this option you tell the compiler not to generate code for channel vectors and channel context.

Related information

-

C compiler option: `--no-warnings (-w)`

Menu entry

1. Select **C Compiler » Diagnostics**.

The Suppress C compiler warnings box shows the warnings that are currently suppressed.

2. To suppress a warning, click on the **Add** button in the **Suppress warnings** box.
3. Enter the numbers, separated by commas or as a range, of the warnings you want to suppress (for example 537, 538). Or you can use the **Add** button multiple times.
4. To suppress all warnings, enable the option **Suppress all warnings**.

Use the **Edit** and **Delete** button to change a warning number or to remove a number from the list.

Command line syntax

```
--no-warnings [=number[-number], ...]
```

```
-w [number[-number], ...]
```

Description

With this option you can suppresses all warning messages or specific warning messages.

On the command line this option works as follows:

- If you do not specify this option, all warnings are reported.
- If you specify this option but without numbers, all warnings are suppressed.
- If you specify this option with a number or a range, only the specified warnings are suppressed. You can specify the option `--no-warnings=number` multiple times.

Example

To suppress warnings 537 and 538, enter:

```
ccpc test.c --no-warnings=537,538
```

Related information

[C compiler option `--warnings-as-errors`](#) (Treat warnings as errors)

[Pragma warning](#)

C compiler option: **--optimize (-O)**

Menu entry

1. Select **C Compiler » Optimization**.
2. Select an optimization level in the **Optimization level** box.

Command line syntax

`--optimize[=flags]`

`-Oflags`

You can set the following flags:

+/-coalesce	a/A	Coalescer: remove unnecessary moves
+/-ipro	b/B	Interprocedural register optimizations
+/-cse	c/C	Common subexpression elimination
+/-expression	e/E	Expression simplification
+/-flow	f/F	Control flow simplification
+/-glo	g/G	Generic assembly code optimizations
+/-inline	i/I	Automatic function inlining
+/-loop	l/L	Loop transformations
+/-forward	o/O	Forward store
+/-propagate	p/P	Constant propagation
+/-compact	r/R	Code compaction (reverse inlining)
+/-subscript	s/S	Subscript strength reduction
+/-peephole	y/Y	Peephole optimizations

Use the following options for predefined sets of flags:

--optimize=0	-O0	No optimization Alias for -OaBCEFGILOPRSY
---------------------	------------	---

No optimizations are performed except for the coalescer (to allow better debug information). The compiler tries to achieve an optimal 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.

--optimize=1	-O1	Optimize Alias for -OabcefgILOPRSy
---------------------	------------	--

Enables optimizations that do not affect the debug ability of the source code. Use this level when you encounter problems during debugging your source code with optimization level 2.

--optimize=2 **-O2** Optimize more (default)
Alias for **-OabcefgIloprSy**

Enables more optimizations to reduce code size and/or execution time. This is the default optimization level.

--optimize=3 **-O3** Optimize most
Alias for **-OabcefgiloprSy**

This is the highest optimization level. Use this level to decrease execution time to meet your real-time requirements.

Default: **--optimize=2**

Description

With this option you can control the level of optimization. If you do not use this option, the default optimization level is *Optimize more* (option **--optimize=2** or **--optimize**).

When you use this option to specify a set of optimizations, you can override these settings in your C source file with `#pragma optimize flag/#pragma endoptimize`.

In addition to the option **--optimize**, you can specify the option **--tradeoff (-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 optimization set:

```
cpcp test.c
```

```
cpcp --optimize=2 test.c
cpcp -O2 test.c
```

```
cpcp --optimize test.c
cpcp -O test.c
```

```
cpcp -OabcefgIloprSy test.c
cpcp --optimize=+coalesce,+ipro,+cse,+expression,+flow,
      +glo,-inline,+loop,+forward,+propagate,
      +compact,+subscript,+peephole test.c
```

Related information

[C compiler option **--tradeoff**](#) (Trade off between speed and size)

[Pragma `optimize/endoptimize`](#)

[Section 3.6, *Compiler Optimizations*](#)

C compiler option: --option-file (-f)

Menu entry

1. Select **C Compiler » Miscellaneous**.
2. Add the option **--option-file** to the **Additional options** field.

Be aware that the options in the option file are added to the C compiler options you have set in the other pages. Only in extraordinary cases you may want to use them in combination.

Command line syntax

`--option-file=file,...`

`-f file,...`

Description

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

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

You can specify the option **--option-file** multiple times.

Format of an option file

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

```
--debug-info  
--define=DEMO=1  
test.c
```

Specify the option file to the compiler:

```
cpcp --option-file=myoptions
```

This is equivalent to the following command line:

```
cpcp --debug-info --define=DEMO=1 test.c
```

Related information

-

C compiler option: --output (-o)

Menu entry

Eclipse names the output file always after the C source file.

Command line syntax

`--output=file`

`-o file`

Description

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

Example

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

```
cpcp --output=output.src test.c
```

Related information

-

C compiler option: `--preprocess (-E)`

Menu entry

1. Select **C Compiler » Preprocessing**.
2. Enable the option **Store preprocessor output in <file>.pre**.
3. (Optional) Enable the option **Keep comments in preprocessor output**.
4. (Optional) Enable the option **Keep #line info in preprocessor output**.

Command line syntax

`--preprocess [=flags]`

`-E[flags]`

You can set the following flags:

+/-comments	c/C	keep comments
+/-includes	i/I	generate a list of included source files
+/-list	l/L	generate a list of macro definitions
+/-make	m/M	generate dependencies for make
+/-noline	p/P	strip #line source position information

Default: `-ECILMP`

Description

With this option you tell the compiler to preprocess the C source. Under Eclipse the compiler sends the preprocessed output to the file `name.pre` (where `name` is the name of the C source file to compile). Eclipse also compiles the C source.

On the command line, the compiler sends the preprocessed file to stdout. To capture the information in a file, specify an output file with the option `--output`.

With `--preprocess=+comments` you tell the preprocessor to keep the comments from the C source file in the preprocessed output.

With `--preprocess=+includes` the compiler will generate a list of all included source files. The preprocessor output is discarded.

With `--preprocess=+list` the compiler will generate a list of all macro definitions. The preprocessor output is discarded.

With `--preprocess=+make` the compiler will generate 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 extension `.o`. With the option `--make-target` you can specify a target name which overrules the default target name.

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

Example

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

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

Related information

[C compiler option --dep-file](#) (Generate dependencies in a file)

[C compiler option --make-target](#) (Specify target name for **-Em** output)

C compiler option: `--preserve-r7-flags`

Menu entry

1. Select **Global Options » Channel Configuration**.
2. Enable the option **Preserve R7 flags 0..7**.

Command line syntax

`--preserve-r7-flags`

Description

With this option the R7 flags are preserved for PRAM access. When accessing the PRAM, which requires loading of R7.DPTR, the IEN and CEN flags in register R7 are preserved. By default the R7 flag registers are not preserved, all flags are cleared for each PRAM access.

Related information

C compiler option `--interrupt-enable` (enable interrupts)

C compiler option: `--rename-sections (-R)`

Menu entry

1. Select **C Compiler » Miscellaneous**.
2. Add the option `--rename-sections` to the **Additional options** field.

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 types you can use are: data and code. 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 use the pragmas `section/endsection` in the C source.

Example

To rename sections of memory type data to `.pcpdata.cpcp_test_variable_name`:

```
cpcp --rename-sections=data=cpcp_{module}_{name} test.c
```

Related information

See [assembler directive `.SDECL`](#) for a list of section types and attributes.

[Pragmas `section/endsection`](#)

[Section 1.10, *Compiler Generated Sections*](#)

C compiler option: `--signed-bitfields`

Menu entry

1. Select **C Compiler » Language**.
2. Enable the option **Treat "int" bit-fields as signed**.

Command line syntax

`--signed-bitfields`

Description

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

Related information

[Section 1.1, *Data Types*](#)

C compiler option: --silicon-bug

Menu entry

1. Expand **C/C++ Build** and select **Processor**.
2. From the **Processor Selection** list, select a processor.

*The **CPU Problem Bypasses and Checks** box shows the available workarounds/checks available for the selected processor.*

3. (Optional) Select **Show all CPU problem bypasses and checks**.
4. Click **Select All** or select one or more individual options.

Command line syntax

`--silicon-bug=arg,...`

You can give the following argument:

pcp-tc038	Workaround for CPU_TC.038
------------------	---------------------------

Description

With this option you specify for which hardware problems the compiler should generate workarounds. Please refer to [Chapter 13, CPU Problem Bypasses and Checks](#) for more information about the individual problems and workarounds.

Example

To enable workarounds for problem PCP_TC.038, enter:

```
cpcp --silicon-bug=pcp-tc038 test.c
```

Related information

[Chapter 13, CPU Problem Bypasses and Checks](#)

Assembler option [--silicon-bug](#)

C compiler option: `--source (-s)`

Menu entry

1. Select **C Compiler » Miscellaneous**.
2. Enable the option **Merge C source code with generated assembly**.

Command line syntax

`--source`

`-s`

Description

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

Related information

Pragmas [source/nosource](#)

C compiler option: `--static`

Menu entry

-

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.

To overrule this option for a specific function or variable, you can use the `export` attribute. For example, when a variable is accessed from assembly:

```
int i __attribute__((export)); /* 'i' has external linkage */
```

With the `export` attribute the compiler will not perform optimizations that affect the unknown code.

Example

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

Related information

-

C compiler option: `--stdout (-n)`

Menu entry

-

Command line syntax

`--stdout`

`-n`

Description

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

Related information

-

C compiler option: `--symbol-prefix`

Menu entry

1. Select **Global Options » Channel Configuration**.
2. Enter a **Prefix for global symbols**.

Command line syntax

`--symbol-prefix=name`

Default: `--symbol-prefix=_PCP`

Description

With this option you can define what prefix is used for global symbols. When you link a TriCore application with a PCP application it is required to prefix the PCP global symbols to avoid duplicate name conflicts between the TriCore and PCP application parts. By default global symbols are prefixed with `_PCP`. When you link multiple PCP channels separately it is required that each channel uses its own global symbol prefix.

Note that the compiler adds an extra underscore to this prefix. For example, the function `main` will get the symbol name `_PCP_main`.

Related information

Assembler option [--symbol-prefix](#)

C compiler option: `--tradeoff (-t)`

Menu entry

1. Select **C Compiler » Optimization**.
2. Select a trade-off level in the **Trade-off between speed and size** box.

Command line syntax

```
--tradeoff={0|1|2|3|4}
```

```
-t{0|1|2|3|4}
```

Default: `--tradeoff=4`

Description

If the compiler uses certain optimizations (option `--optimize`), 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).

By default the compiler optimizes for code size (`--tradeoff=4`).

If you have not specified the option `--optimize`, the compiler uses the default *Optimize more* optimization. In this case it is still useful to specify a trade-off level.

Example

To set the trade-off level for the used optimizations:

```
cpcp --tradeoff=2 test.c
```

The compiler uses the default *Optimize more* optimization level and balances speed and size while optimizing.

Related information

[C compiler option `--optimize`](#) (Specify optimization level)

[Section 3.6.3, *Optimize for Size or Speed*](#)

C compiler option: `--uchar (-u)`

Menu entry

1. Select **C Compiler » Language**.
2. Enable the option **Treat "char" variables as unsigned**.

Command line syntax

`--uchar`

`-u`

Description

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

Related information

Section 1.1, *Data Types*

C compiler option: `--undefine (-U)`

Menu entry

1. Select **C Compiler » Preprocessing**

The Defined symbols box shows the symbols that are currently defined.

2. To remove a defined symbol, select the symbol in the **Defined symbols** box and click on the **Delete** button.

Command line syntax

`--undefine=macro_name`

`-Umacro_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:

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

Example

To undefine the predefined macro `__TASKING__`:

```
cpcp --undefine=__TASKING__ test.c
```

Related information

C compiler option `--define` (Define preprocessor macro)

Section 1.6, *Predefined Preprocessor Macros*

C compiler option: --version (-V)

Menu entry

-

Command line syntax

`--version`

`-v`

Description

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

Example

```
cpcp --version
```

The compiler does not compile any files but displays the following version information:

```
TASKING VX-toolset for PCP: C compiler    vx.yrz Build nnn  
Copyright 2006-year Altium BV           Serial# 00000000
```

Related information

-

C compiler option: `--warnings-as-errors`

Menu entry

1. Select **Global Options**.
2. Enable the option **Treat warnings as errors**.

Command line syntax

```
--warnings-as-errors [=number [-number] , ...]
```

Description

If the compiler encounters an error, it stops compiling. When you use this option without arguments, you tell the compiler to treat all warnings not suppressed by option `--no-warnings` (or `#pragma warning`) 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 or ranges. In this case, this option takes precedence over option `--no-warnings` (and `#pragma warning`).

Related information

[C compiler option `--no-warnings`](#) (Suppress some or all warnings)

[Pragma warning](#)

8.2. Assembler Options

This section lists all assembler options.

Options in Eclipse versus options on the command line

Most command line options have an equivalent option in Eclipse but some options are only available on the command line. Eclipse invokes the assembler via the control program. Therefore, it uses the syntax of the control program to pass options and files to the assembler. If there is no equivalent option in Eclipse, you can specify a command line option in Eclipse as follows:

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

The Properties dialog appears.

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

In the right pane the Settings appear.

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

4. In the **Additional options** field, enter one or more command line options.

*Because Eclipse uses the control program, Eclipse automatically precedes the option with **-Wa** to pass the option via the control program directly to the assembler.*

Note that the options you enter in the Assembler page are not only used for hand-coded assembly files, but also for the assembly files generated by the compiler.

Be aware that some command line options are not useful in Eclipse or just do not have any effect. For example, the option **-V** displays version header information and has no effect in Eclipse.

Short and long option names

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

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

```
aspcp -Ogs test.src
aspcp --optimize+=generics,+instr-size test.src
```

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

Assembler option: `--case-insensitive (-c)`

Menu entry

1. Select **Assembler » Symbols**.
2. Enable the option **Case insensitive identifiers**.

Command line syntax

`--case-insensitive`

`-c`

Default: case sensitive

Description

With this option you tell the assembler not to distinguish between upper and lower case characters. By default the assembler considers upper and lower case characters as different characters.

Assembly source files that are generated by the compiler must always be assembled case sensitive. When you are writing your own assembly code, you may want to specify the case insensitive mode.

Example

When assembling case insensitive, the label `LabelName` is the same label as `labelname`.

```
aspcp --case-insensitive test.src
```

Related information

Assembler control `$CASE`

Assembler option: **--check**

Menu entry

-

Command line syntax

--check

Description

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

The assembler reports any warnings and/or errors.

This option is available on the command line only.

Related information

[C compiler option **--check**](#) (Check syntax)

Assembler option: `--core`

Menu entry

1. Select **Assembler » Miscellaneous**.
2. Add the option `--core` to the **Additional options** field.

Command line syntax

`--core=core`

You can specify the following *core* arguments:

pcp1	PCP 1 syntax
pcp2	PCP 2 syntax

Default: derived from `--cpu`

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.

For more information see assembler option `--cpu`.

Example

Specify a custom core:

```
aspcp --core=pcp2 test.asm
```

Related information

[Assembler option `--cpu`](#) (Select processor)

Assembler option: **--cpu (-C)**

Menu entry

1. Expand **C/C++ Build** and select **Processor**.
2. From the **Processor Selection** list, select a processor or select **User defined TriCore**

Command line syntax

`--cpu=cpu`

`-Ccpu`

Description

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

Based on this option the assembler always includes the special function register file `regcpu.def`, unless you disable the option **Automatic inclusion of '.def' file** on the **Preprocessing** page ([option --no-tasking-sfr](#)).

To avoid conflicts, make sure you specify the same target processor as you did for the compiler (Eclipse and the control program do this automatically).

Example

To assemble the file `test.asm` for the TC1165 processor and use the register file `regtc1165.def`:

```
aspcp --cpu=tc1165 test.asm
```

Related information

[Assembler option --core](#) (Select the core)

[Assembler option --no-tasking-sfr](#) (Do not include .def file)

[Section 2.5.1, Special Function Registers](#)

Assembler option: `--debug-info (-g)`

Menu entry

1. Select **Assembler » Symbols**.
2. Select an option from the **Generate symbolic debug** list.

Command line syntax

`--debug-info[=flags]`

`-g[flags]`

You can set the following flags:

+/-asm	a/A	Assembly source line information
+/-hll	h/H	Pass high level language debug information (HLL)
+/-local	l/L	Assembler local symbols debug information
+/-smart	s/S	Smart debug information

Default: `--debug-info=+hll`

Default (without flags): `--debug-info=+smart`

Description

With this option you tell the assembler which kind of debug information to emit in the object file.

You cannot specify `--debug-info=+asm,+hll`. Either the assembler generates assembly source line information, or it passes HLL debug information.

When you specify `--debug-info=+smart`, 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 `--debug-info=-asm,+hll,-local`). If not, the assembler generates assembly source line information (same as `--debug-info=+asm,-hll,+local`).

With `--debug-info=AHLS` the assembler does not generate any debug information.

Related information

Assembler control `$DEBUG`

Assembler option: --define (-D)

Menu entry

1. Select **Assembler » Preprocessing**.

The Defined symbols box right-below shows the symbols that are currently defined.

2. To define a new symbol, click on the **Add** button in the **Defined symbols** box.
3. Type the symbol definition (for example, demo=1)

Use the **Edit** and **Delete** button to change a macro definition or to remove a macro from the list.

Command line syntax

```
--define=macro_name[=macro_definition]
```

```
-Dmacro_name[=macro_definition]
```

Description

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

You can specify as many macros as you like. Simply use the **Add** button to add new macro definitions.

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

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

This option has the same effect as defining symbols via the `.DEFINE`, `.SET`, and `.EQU` directives. (similar to `#define` in the C language). With the `.MACRO` directive you can define more complex macros.

Example

Consider the following assembly program with conditional code to assemble a demo program and a real program:

```
.IF DEMO == 1
...           ; instructions for demo application
.ELSE
...           ; instructions for the real application
.ENDIF
```

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

```
aspcp --define=DEMO test.src  
aspcp --define=DEMO=1 test.src
```

Note that both invocations have the same effect.

Related information

[Assembler option **--option-file**](#) (Specify an option file)

Assembler option: `--diag`

Menu entry

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

The Problems view is added to the current perspective.

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

A popup menu appears.

3. Select **Detailed Diagnostics Info**.

A dialog box appears with additional information.

Command line syntax

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

You can set the following output formats:

html	HTML output.
rtf	Rich Text Format.
text	ASCII text.

Default format: text

Description

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

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

Example

To display an explanation of message number 244, enter:

```
aspcp --diag=244
```

This results in the following message and explanation:

```
w244: additional input files will be ignored
```

The assembler supports only a single input file. All other input files are ignored.

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

```
aspcp --diag=html:all > aserrors.html
```

Related information

[Section 4.6, *Assembler Error Messages*](#)

Assembler option: `--emit-locals`

Menu entry

1. Select **Assembler » Symbols**.
2. Enable or disable one or both of the following options:
 - Emit local EQU symbols
 - Emit local non-EQU symbols

Command line syntax

`--emit-locals[=flag,...]`

You can set the following flags:

<code>+/-equs</code>	<code>e/E</code>	emit local EQU symbols
<code>+/-symbols</code>	<code>s/S</code>	emit local non-EQU symbols

Default: `--emit-locals=ES`

Default (without flags): `--emit-locals=+symbols`

Description

With the option `--emit-locals=+equs` the assembler also emits local EQU symbols to the object file. Normally, only global symbols and non-EQU local symbols are emitted. Having local symbols in the object file can be useful for debugging.

Related information

Assembler directive [.EQU](#)

Assembler option: `--error-file`

Menu entry

-

Command line syntax

```
--error-file[=file]
```

Description

With this option the assembler redirects error messages to a file. If you do not specify a filename, the error file will be named after the input file with extension `.ers`.

Example

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

```
aspcp --error-file=errors.ers test.src
```

Related information

[Section 4.6, *Assembler Error Messages*](#)

Assembler option: **--error-limit**

Menu entry

1. Select **Assembler » Diagnostics**.
2. Enter a value in the **Maximum number of emitted errors** field.

Command line syntax

`--error-limit=number`

Default: 42

Description

With this option you tell the assembler to only emit the specified maximum number of errors. When 0 (null) is specified, the assembler emits all errors. Without this option the maximum number of errors is 42.

Related information

[Section 4.6, *Assembler Error Messages*](#)

Assembler option: --help (-?)

Menu entry

-

Command line syntax

```
--help[=item]
```

```
-?
```

You can specify the following arguments:

options	Show extended option descriptions
----------------	-----------------------------------

Description

Displays an overview of all command line options. When you specify the argument **options** you can list detailed option descriptions.

Example

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

```
aspcp -?  
aspcp --help  
aspcp
```

To see a detailed description of the available options, enter:

```
aspcp --help=options
```

Related information

-

Assembler option: --include-directory (-I)

Menu entry

1. Select **Assembler » Include Paths**.

The Include paths box shows the directories that are added to the search path for include files.

2. To define a new directory for the search path, click on the **Add** button in the **Include paths** box.
3. Type or select a path.

Use the **Edit** and **Delete** button to change a path or to remove a path from the list.

Command line syntax

```
--include-directory=path,...
```

```
-Ipath,...
```

Description

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

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

1. The pathname in the assembly file and the directory of the assembly source.
2. The path that is specified with this option.
3. The path that is specified in the environment variable `ASPCPINC` when the product was installed.
4. The default directory `$(PRODDIR)\include`.

Example

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

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

You can call the assembler as follows:

```
aspcp --include-directory=c:\proj\include test.src
```

First the assembler looks for the file `myinc.inc` in the directory where `test.src` is located. If it does not find the file, it looks in the directory `c:\proj\include` (this option). If the file is still not found, the assembler searches in the environment variable and then in the default include directory.

Related information

Assembler option **--include-file** (Include file at the start of the input file)

Assembler option: --include-file (-H)

Menu entry

1. Select **Assembler » Preprocessing**.

The Pre-include files box shows the files that are currently included before the compilation starts.

2. To define a new file, click on the **Add** button in the **Pre-include files** box.
3. Type the full path and file name or select a file.

Use the **Edit** and **Delete** button to change a file name or to remove a file from the list.

Command line syntax

```
--include-file=file,...
```

```
-Hfile,...
```

Description

With this option (set at project level) you include one extra file at the beginning of the assembly source file. The specified include file is included before all other includes. This is the same as specifying `.INCLUDE 'file'` at the beginning of your assembly source.

Example

```
aspcp --include-file=myinc.inc test.src
```

The file `myinc.inc` is included at the beginning of `test.src` before it is assembled.

Related information

Assembler option **--include-directory** (Add directory to include file search path)

Assembler option: **--keep-output-files (-k)**

Menu entry

Eclipse *always* removes the object file when errors occur during assembling.

Command line syntax

--keep-output-files

-k

Description

If an error occurs during assembling, the resulting object file (.o) may be incomplete or incorrect. With this option you keep the generated object file when an error occurs.

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

Use this option when you still want to use the generated object. For example when you know that a particular error does not result in a corrupt object file.

Related information

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

Assembler option: **--list-file (-l)**

Menu entry

1. Select **Assembler » List File**.
2. Enable the option **Generate list file**.
3. Enable or disable the types of information to be included.

Command line syntax

`--list-file[=file]`

`-l[file]`

Default: no list file is generated

Description

With this option you tell the assembler to generate a list file. A list file shows the generated object code and the relative addresses. Note that the assembler generates a relocatable object file with relative addresses.

With the optional *file* you can specify an alternative name for the list file. By default, the name of the list file is the basename of the source file with the extension `.lst`.

Related information

Assembler option [--list-format](#) (Format list file)

Assembler option: --list-format (-L)

Menu entry

1. Select **Assembler » List File**.
2. Enable the option **Generate list file**.
3. Enable or disable the types of information to be included.

Command line syntax

`--list-format=flag,...`

`-Lflags`

You can set the following flags:

+/-section	d/D	List section directives (.SDECL, .SECT)
+/-symbol	e/E	List symbol definition directives
+/-generic-expansion	g/G	List expansion of generic instructions
+/-generic	i/I	List generic instructions
+/-macro	m/M	List macro definitions
+/-empty-line	n/N	List empty source lines (newline)
+/-conditional	p/P	List conditional assembly
+/-equate	q/Q	List equate and set directives (.EQU, .SET)
+/-relocations	r/R	List relocations characters 'r'
+/-equate-values	v/V	List equate and set values
+/-wrap-lines	w/W	Wrap source lines
+/-macro-expansion	x/X	List macro expansions
+/-cycle-count	y/Y	List cycle counts
+/-define-expansion	z/Z	List define expansions

Use the following options for predefined sets of flags:

<code>--list-format=0</code>	-L0	All options disabled Alias for <code>--list-format=DEGIMNPQRVWXYZ</code>
<code>--list-format=1</code>	-L1	All options enabled Alias for <code>--list-format=degimnpqrwxyz</code>

Default: `--list-format=dEGiMnPqrVwXyZ`

Description

With this option you specify which information you want to include in the list file.

On the command line you must use this option in combination with the option **--list-file (-l)**.

Related information

Assembler option **--list-file** (Generate list file)

Assembler option **--section-info=**+list**** (Display section information in list file)

Assembler option: `--no-tasking-sfr`

Menu entry

1. Select **Assembler » Preprocessing**.
2. Disable the option **Automatic inclusion of '.def' file**.

Command line syntax

```
--no-tasking-sfr
```

Description

Normally, the assembler includes a special function register (SFR) file before assembling. The assembler automatically selects the SFR file belonging to the target you select on the **Processor** page (assembler option `--cpu`).

With this option the assembler does not include the register file `regcpu.def` as based on the selected target processor.

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

Example

```
aspcp --cpu=tc1165 --no-tasking-sfr test.src
```

The register file `regtc1165.def` is not included.

Related information

[Assembler option `--cpu`](#) (Select processor)

Assembler option: --no-warnings (-w)

Menu entry

1. Select **Assembler » Diagnostics**.

The Suppress warnings box shows the warnings that are currently suppressed.

2. To suppress a warning, click on the **Add** button in the **Suppress warnings** box.
3. Enter the numbers, separated by commas, of the warnings you want to suppress (for example 201, 202). Or you can use the **Add** button multiple times.
4. To suppress all warnings, enable the option **Suppress all warnings**.

Use the **Edit** and **Delete** button to change a warning number or to remove a number from the list.

Command line syntax

`--no-warnings[=number, ...]`

`-w[number, ...]`

Description

With this option you can suppresses all warning messages or specific warning messages.

On the command line this option works as follows:

- If you do not specify this option, all warnings are reported.
- If you specify this option but without numbers, all warnings are suppressed.
- If you specify this option with a number, only the specified warning is suppressed. You can specify the option `--no-warnings=number` multiple times.

Example

To suppress warnings 201 and 202, enter:

```
aspcp test.src --no-warnings=201,202
```

Related information

[Assembler option --warnings-as-errors](#) (Treat warnings as errors)

Assembler option: `--optimize (-O)`

Menu entry

1. Select **Assembler » Optimization**.
2. Select one or more of the following options:
 - Optimize generic instructions
 - Optimize instruction size

Command line syntax

`--optimize=flag,...`

`-Oflags`

You can set the following flags:

+/-generics	g/G	Allow generic instructions
+/-instr-size	s/S	Optimize instruction size

Default: `--optimize=gs`

Description

With this option you can control the level of optimization. For details about each optimization see [Section 4.4, *Assembler Optimizations*](#).

Related information

Assembler control `$HW_ONLY`

[Section 4.4, *Assembler Optimizations*](#)

Assembler option: --option-file (-f)

Menu entry

1. Select **Assembler » Miscellaneous**.
2. Add the option **--option-file** to the **Additional options** field.

Be aware that the options in the option file are added to the assembler options you have set in the other pages. Only in extraordinary cases you may want to use them in combination.

Command line syntax

```
--option-file=file,...
```

```
-f file,...
```

Description

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

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

Option files can also be generated on the fly, for example by the make utility. You can specify the option **--option-file** multiple times.

Format of an option file

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

```
--debug=+asm,-local  
test.src
```

Specify the option file to the assembler:

```
aspcp --option-file=myoptions
```

This is equivalent to the following command line:

```
aspcp --debug=+asm,-local test.src
```

Related information

-

Assembler option: **--output (-o)**

Menu entry

Eclipse names the output file always after the input file.

Command line syntax

```
--output=file
```

```
-o file
```

Description

With this option you can specify another filename for the output file of the assembler. Without this option, the basename of the assembly source file is used with extension `.o`.

Example

To create the file `relobj.o` instead of `asm.o`, enter:

```
aspcp --output=relobj.o asm.src
```

Related information

-

Assembler option: `--page-length`

Menu entry

1. Select **Assembler » Miscellaneous**.
2. Add the option `--page-length` to the **Additional options** field.

Command line syntax

`--page-length=number`

Default: 72

Description

If you generate a list file with the assembler option `--list-file`, this option sets the number of lines in a page in the list file. The default is 72, the minimum is 10. As a special case, a page length of 0 turns off page breaks.

Related information

[Assembler option `--list-file`](#) (Generate list file)

[Assembler control `\$PAGE`](#)

Assembler option: **--page-width**

Menu entry

1. Select **Assembler » Miscellaneous**.
2. Add the option **--page-width** to the **Additional options** field.

Command line syntax

--page-width=*number*

Default: 132

Description

If you generate a list file with the assembler option **--list-file**, this option sets the number of columns per line on a page in the list file. The default is 132, the minimum is 40.

Related information

Assembler option **--list-file** (Generate list file)

Assembler control **\$PAGE**

Assembler option: `--preprocess (-E)`

Menu entry

-

Command line syntax

`--preprocess`

`-E`

Description

With this option the assembler will only preprocess the assembly source file. The assembler sends the preprocessed file to stdout.

Related information

-

Assembler option: `--preprocessor-type (-m)`

Menu entry

-

Command line syntax

`--preprocessor-type=type`

`-mtype`

You can set the following preprocessor types:

none	n	No preprocessor
tasking	t	TASKING preprocessor

Default: `--preprocessor-type=tasking`

Description

With this option you select the preprocessor that the assembler will use. By default, the assembler uses the TASKING preprocessor.

When the assembly source file does not contain any preprocessor symbols, you can specify to the assembler not to use a preprocessor.

Related information

-

Assembler option: `--section-info (-t)`

Menu entry

1. Select **Assembler** » **List File**.
2. Enable the option **Generate list file**.
3. Enable the option **List section summary**.

and/or

1. Select **Assembler** » **Diagnostics**.
2. Enable the option **Display section summary**.

Command line syntax

```
--section-info[=flag,...]
```

```
-t[flags]
```

You can set the following flags:

+/-console	c/C	Display section summary on console
+/-list	I/L	List section summary in list file

Default: `--section-info=CL`

Default (without flags): `--section-info=cI`

Description

With this option you tell the assembler to display section information. For each section its memory space, size, total cycle counts and name is listed on stdout and/or in the list file.

The cycle count consists of two parts: the total accumulated count for the section and the total accumulated count for all repeated instructions. In the case of nested loops it is possible that the total supersedes the section total.

Example

To writes the section information to the list file and also display the section information on stdout, enter:

```
aspcp --list-file --section-info asm.src
```

Related information

Assembler option `--list-file` (Generate list file)

Assembler option: `--silicon-bug`

Menu entry

1. Expand **C/C++ Build** and select **Processor**.
2. From the **Processor Selection** list, select a processor.

*The **CPU Problem Bypasses and Checks** box shows the available workarounds/checks available for the selected processor.*

3. (Optional) Select **Show all CPU problem bypasses and checks**.
4. Click **Select All** or select one or more individual options.

Command line syntax

`--silicon-bug=arg,...`

You can give one or more of the following arguments:

pcp-tc034	Check for PCP_TC.034
pcp-tc038	Check for PCP_TC.038

Description

With this option you specify for which hardware problems the assembler should check or generate workarounds. Please refer to [Chapter 13, CPU Problem Bypasses and Checks](#) for more information about the individual problems and workarounds.

Example

To check for problems PCP_TC.034 and PCP_TC.038, enter:

```
aspcp --silicon-bug=pcp-tc034,pcp-tc038 test.src
```

Related information

[Chapter 13, CPU Problem Bypasses and Checks](#)

C compiler option `--silicon-bug`

Assembler option: `--symbol-prefix (-P)`

Menu entry

1. Select **Assembler** » **Miscellaneous**.
2. Add the option `--symbol-prefix` to the **Additional options** field.

Command line syntax

```
--symbol-prefix=prefix
```

```
-Pprefix
```

Description

With this option you can specify a prefix to use for global and external symbols. When you link a TriCore application with a PCP application it is required to prefix the PCP global symbols to avoid duplicate name conflicts between the TriCore and PCP application parts. When you link multiple PCP channels separately it is required that each channel uses its own global symbol prefix.

Note that the C compiler by default adds the prefix `_PCP_`.

Example

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

```
aspcp --symbol-prefix=_PCP_ test.asm
```

Related information

[C compiler option `--symbol-prefix`](#)

Assembler option: `--symbol-scope (-i)`

Menu entry

1. Select **Assembler » Symbols**.
2. Enable or disable the option **Set default symbol scope to global**.

Command line syntax

`--symbol-scope=scope`

`-i scope`

You can set the following scope:

global	g	Default symbol scope is global
local	l	Default symbol scope is local

Default: `--symbol-scope=local`

Description

With this option you tell the assembler how to treat symbols that you have not specified explicitly as global or local. By default the assembler treats all symbols as local symbols unless you have defined them explicitly as global.

Related information

Assembler directive [.GLOBAL](#)

Assembler directive [.LOCAL](#)

Assembler control [\\$IDENT](#)

Assembler option: --version (-V)

Menu entry

-

Command line syntax

`--version`

`-v`

Description

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

Example

```
aspcp --version
```

The assembler does not assemble any files but displays the following version information:

```
TASKING VX-toolset for PCP: PCP assembler    vx.yrz Build nnn  
Copyright 2002-year Altium BV              Serial# 00000000
```

Related information

-

Assembler option: **--warnings-as-errors**

Menu entry

1. Select **Global Options**.
2. Enable the option **Treat warnings as errors**.

Command line syntax

--warnings-as-errors [=number, ...]

Description

If the assembler encounters an error, it stops assembling. When you use this option without arguments, you tell the assembler to treat all warnings as errors. This means that the exit status of the assembler will be non-zero after one or more assembler warnings. As a consequence, the assembler now also stops after encountering a warning.

You can also limit this option to specific warnings by specifying a comma-separated list of warning numbers.

Related information

Assembler option **--no-warnings** (Suppress some or all warnings)

8.3. Linker Options

This section lists all linker options.

Options in Eclipse versus options on the command line

Most command line options have an equivalent option in Eclipse but some options are only available on the command line. Eclipse invokes the linker via the control program. Therefore, it uses the syntax of the control program to pass options and files to the linker. If there is no equivalent option in Eclipse, you can specify a command line option in Eclipse as follows:

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

The Properties dialog appears.

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

In the right pane the Settings appear.

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

4. In the **Additional options** field, enter one or more command line options.

*Because Eclipse uses the control program, Eclipse automatically precedes the option with **-WI** to pass the option via the control program directly to the linker.*

Be aware that some command line options are not useful in Eclipse or just do not have any effect. For example, the option **--keep-output-files** keeps files after an error occurred. When you specify this option in Eclipse, it will have no effect because Eclipse always removes the output file after an error had occurred.

Short and long option names

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

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

```
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.

Linker option: Include debugger synchronization utility

Menu entry

1. Select **Linker » Miscellaneous**.
2. Enable the option **Include debugger synchronization utility**.

Command line syntax

```
--extern=_PCP_sync_on_halt
```

Description

When the debugger stops the TriCore, this does not automatically flush all the states in the CPU's pipeline and caches. In order to be able to correctly show the program state, the debugger therefore needs to execute special flushing code every time the CPU halts. When you enable the option **Include debugger synchronization utility** this causes extra code (`_PCP_sync_on_halt` and other symbols) to be linked in. If you are not going to use the debugger, you can save a few tens of bytes by disabling this option.

Related information

[Linker option --extern](#)

Linker option: `--case-insensitive`

Menu entry

1. Select **Linker » Miscellaneous**.
2. Enable the option **Link case insensitive**.

Command line syntax

`--case-insensitive`

Default: case sensitive

Description

With this option you tell the linker not to distinguish between 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.

Related information

Assembler option `--case-insensitive`

Linker option: `--chip-output (-c)`

Menu entry

1. Select **Linker » Output Format**.
2. Enable the option **Generate Intel Hex format file** and/or **Generate S-records file**.
3. Enable the option **Create file for each memory chip**.
4. Optionally, specify the **Size of addresses**.

Eclipse always uses the project name as the basename for the output file.

Command line syntax

```
--chip-output=[basename]:format[:addr_size],...
```

```
-c[basename]:format[:addr_size],...
```

You can specify the following formats:

IHEX	Intel Hex
SREC	Motorola S-records

The *addr_size* specifies the size of the addresses in bytes (record length). For Intel Hex you can use the values **1**, **2** or **4** bytes (default). For Motorola-S you can specify: **2** (S1 records), **3** (S2 records) or **4** bytes (S3 records, default).

Description

With this option you specify the Intel Hex or Motorola S-record output format for loading into a PROM-programmer. The linker generates a file for each ROM memory defined in the LSL file, where sections are located:

```
memory memname
{ type=rom; }
```

The name of the file is the name of the Eclipse project or, on the command line, the name of the memory device that was emitted with extension `.hex` or `.sre`. Optionally, you can specify a *basename* which prepends the generated file name.

The linker always outputs a debugging file in ELF/DWARF format and optionally an absolute object file in Intel Hex-format and/or Motorola S-record format.

Example

To generate Intel Hex output files for each defined memory, enter the following on the command line:

```
lpcp --chip-output=myfile:IHEX test1.o
```

In this case, this generates the file `myfile_memname.hex`.

Related information

Linker option `--output` (Output file)

Linker option: **--define (-D)**

Menu entry

1. Select **Linker » Script File**.

The Defined symbols box shows the symbols that are currently defined.

2. To define a new symbol, click on the **Add** button in the **Defined symbols** box.
3. Type the symbol definition (for example, `demo=1`)

Use the **Edit** and **Delete** button to change a macro definition or to remove a macro from the list.

Command line syntax

```
--define=macro_name[=macro_definition]
```

```
-Dmacro_name[=macro_definition]
```

Description

With this option you can define a macro and specify it to the linker LSL file preprocessor. If you only specify a macro name (no macro definition), the macro expands as '1'.

You can specify as many macros as you like; just use the option **--define (-D)** multiple times. If the command line exceeds the limit of the operating system, you can define the macros in an option file which you then must specify to the linker with the option **--option-file (-f) file**.

The definition can be tested by the preprocessor with `#if`, `#ifdef` and `#ifndef`, for conditional locating.

Example

To define the RESET vector, which is used in the linker script file `tc1v1_3.lsl`, enter:

```
lpcp test.o -otest.elf --lsl-file=tc1v1_3.lsl --define=RESET=0xa0000000
```

Related information

[Linker option **--option-file**](#) (Specify an option file)

Linker option: `--diag`

Menu entry

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

The Problems view is added to the current perspective.

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

A popup menu appears.

3. Select **Detailed Diagnostics Info**.

A dialog box appears with additional information.

Command line syntax

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

You can set the following output formats:

html	HTML output.
rtf	Rich Text Format.
text	ASCII text.

Default format: text

Description

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

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

With this option the linker does not link/locate any files.

Example

To display an explanation of message number 106, enter:

```
lpcp --diag=106
```

This results in the following message and explanation:

```
E106: unresolved external: <message>
```

The linker could not resolve all external symbols.

TASKING VX-toolset for PCP User Guide

This is an error when the incremental linking option is disabled.
The <message> indicates the symbol that is unresolved.

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

```
lpcp --diag=html:all > lkerrors.html
```

Related information

[Section 5.10, *Linker Error Messages*](#)

Linker option: `--error-file`

Menu entry

-

Command line syntax

```
--error-file[=file]
```

Description

With this option the linker redirects error messages to a file. If you do not specify a filename, the error file is `lpcp.elk`.

Example

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

```
lpcp --error-file=errors.elk test.o
```

Related information

Section 5.10, *Linker Error Messages*

Linker option: **--error-limit**

Menu entry

1. Select **Linker » Diagnostics**.
2. Enter a value in the **Maximum number of emitted errors** field.

Command line syntax

`--error-limit=number`

Default: 42

Description

With this option you tell the linker to only emit the specified maximum number of errors. When 0 (null) is specified, the linker emits all errors. Without this option the maximum number of errors is 42.

Related information

Section 5.10, *Linker Error Messages*

Linker option: `--extern (-e)`

Menu entry

-

Command line syntax

```
--extern=symbol,...
```

```
-esymbol,...
```

Description

With this option you force the linker to consider the given symbol as an undefined reference. The linker tries to resolve this symbol, either the symbol is defined in an object file or the linker extracts the corresponding symbol definition from a library.

This option is, for example, useful if the startup code is part of a library. Because your own application does not refer to the startup code, you can force the startup code to be extracted by specifying the symbol `_START` as an unresolved external.

Example

Consider the following invocation:

```
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 --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 5.3, *Linking with Libraries*](#)

Linker option: **--first-library-first**

Menu entry

-

Command line syntax

--first-library-first

Description

When the linker processes a library it searches for symbols that are referenced by the objects and libraries processed so far. If the library contains a definition for an unresolved reference the linker extracts the object that contains the definition from the library.

By default the linker processes object files and libraries in the order in which they appear on the command line. If you specify the option **--first-library-first** the linker always tries to take the symbol definition from the library that appears first on the command line before scanning subsequent libraries.

This is for example useful when you are working with a newer version of a library that partially overlaps the older version. Because they do not contain exactly the same functions, you have to link them both. However, when a function is present in both libraries, you may want the linker to extract the most recent function.

Example

Consider the following example:

```
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**](#) (Rescan libraries to solve unresolved externals)

Linker option: `--global-type-checking`

Menu entry

-

Command line syntax

`--global-type-checking`

Description

Use this option when you want the linker to check the types of variable and function references against their definitions, using DWARF 2 or DWARF 3 debug information.

This check should give the same result as the C compiler when you use MIL linking.

Related information

-

Linker option: --help (-?)

Menu entry

-

Command line syntax

```
--help[=item]
```

```
-?
```

You can specify the following arguments:

options	Show extended option descriptions
----------------	-----------------------------------

Description

Displays an overview of all command line options. When you specify the argument **options** you can list detailed option descriptions.

Example

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

```
lpcp -?  
lpcp --help  
lpcp
```

To see a detailed description of the available options, enter:

```
lpcp --help=options
```

Related information

-

Linker option: `--hex-format`

Menu entry

1. Select **Linker » Miscellaneous**.
2. Add the option `--hex-format` to the **Additional options** field.

Command line syntax

`--hex-format=flag,...`

You can set the following flag:

+/-start-address	s/S	Emit start address record
-------------------------	------------	---------------------------

Default: `--hex-format=s`

Description

With this option you can specify to emit or omit the start address record from the hex file.

Related information

[Linker option `--output`](#) (Output file)

Linker option: **--hex-record-size**

Menu entry

1. Select **Linker » Miscellaneous**.
2. Add the option **--hex-record-size** to the **Additional options** field.

Command line syntax

--hex-record-size=*size*

Default: 32

Description

With this option you can set the size (width) of the Intel Hex data records.

Related information

Linker option **--output** (Output file)

Linker option: `--import-object`

Menu entry

1. Select **Linker » Data Objects**.

The Data objects box shows the list of object files that are imported.

2. To add a data object, click on the **Add** button in the **Data objects** box.
3. Type or select a binary file (including its path).

Use the **Edit** and **Delete** button to change a filename or to remove a data object from the list.

Command line syntax

```
--import-object=file,...
```

Description

With this option the linker imports a binary *file* containing raw data and places it in a section. The section name is derived from the filename, in which dots are replaced by an underscore. So, when importing a file called `my.jpg`, a section with the name `my_jpg` is created. In your application you can refer to the created section by using linker labels.

Related information

[Section 5.5, *Importing Binary Files*](#)

Linker option: **--include-directory (-I)**

Menu entry

-

Command line syntax

--include-directory=*path*,...

-I*path*,...

Description

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

The order in which the linker searches for LSL include files is:

1. The pathname in the LSL file and the directory where the LSL file is located (only for #include files that are enclosed in "")
2. The path that is specified with this option.
3. The default directory $\$(PRODDIR)\include.lsl$.

Example

Suppose that your linker script file `mylsl.lsl` contains the following line:

```
#include "myinc.inc"
```

You can call the linker as follows:

```
lpcp --include-directory=c:\proj\include --lsl-file=mylsl.lsl test.o
```

First the linker looks for the file `myinc.inc` in the directory where `mylsl.lsl` is located. If it does not find the file, it looks in the directory `c:\proj\include` (this option). Finally it looks in the directory $\$(PRODDIR)\include.lsl$.

Related information

[Linker option **--lsl-file**](#) (Specify linker script file)

Linker option: --incremental (-r)

Menu entry

-

Command line syntax

`--incremental`

`-r`

Description

Normally the linker links and locates the specified object files. With this option you tell the linker only to link the specified files. The linker creates a linker output file `.out`. You then can link this file again with other object files until you have reached the final linker output file that is ready for locating.

In the last pass, you call the linker without this option with the final linker output file `.out`. The linker will now locate the file.

Example

In this example, the files `test1.o`, `test2.o` and `test3.o` are incrementally linked:

1. `lpcp --incremental test1.o test2.o --output=test.out`

test1.o and test2.o are linked

2. `lpcp --incremental test3.o test.out`

test3.o and test.out are linked, task1.out is created

3. `lpcp task1.out`

task1.out is located

Related information

[Section 5.4, Incremental Linking](#)

Linker option: **--keep-output-files (-k)**

Menu entry

Eclipse *always* removes the output files when errors occurred.

Command line syntax

--keep-output-files

-k

Description

If an error occurs during linking, the resulting output file may be incomplete or incorrect. With this option you keep the generated output files when an error occurs.

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

Use this option when you still want to use the generated file. For example when you know that a particular error does not result in a corrupt object file, or when you want to inspect the output file, or send it to Altium support.

Related information

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

Linker option: `--library (-l)`

Menu entry

1. Select **Linker » Libraries**.

The Libraries box shows the list of libraries that are linked with the project.

2. To add a library, click on the **Add** button in the **Libraries** box.
3. Type or select a library (including its path).
4. Optionally, disable the option **Link default libraries**.

Use the **Edit** and **Delete** button to change a library name or to remove a library from the list.

Command line syntax

`--library=name`

`-lname`

Description

With this option you tell the linker to use system library `libname.a`, where `name` is a string. The linker first searches for system libraries in any directories specified with `--library-directory`, then in the directories specified with the environment variables `LIBTC1V1_3` / `LIBTC1V1_3_1` / `LIBTC1V1_6`, unless you used the option `--ignore-default-library-path`.

Example

To search in the system library `libc.a` (C library):

```
lpcp test.o mylib.a --library=c
```

The linker links the file `test.o` and first looks in library `mylib.a` (in the current directory only), then in the system library `libc.a` to resolve unresolved symbols.

Related information

Linker option `--library-directory` (Additional search path for system libraries)

Section 5.3, *Linking with Libraries*

Linker option: **--library-directory (-L) / --ignore-default-library-path**

Menu entry

1. Select **Linker » Libraries**.

The Library search path box shows the directories that are added to the search path for library files.

2. To define a new directory for the search path, click on the **Add** button in the **Library search path** box.
3. Type or select a path.

Use the **Edit** and **Delete** button to change a path or to remove a path from the list.

Command line syntax

--library-directory=*path*,...

-L*path*,...

--ignore-default-library-path

-L

Description

With this option you can specify the path(s) where your system libraries, specified with the option **--library (-l)**, are located. If you want to specify multiple paths, use the option **--library-directory** for each separate path.

The default path is $\$(PRODDIR)\lib\[pcp1][pcp2]$.

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_3 / LIBTC1V1_3_1 / LIBTC1V1_6`. So, the linker ignores steps 2 and 3 as listed below.

The priority order in which the linker searches for system libraries specified with the option **--library (-l)** is:

1. The path that is specified with the option **--library-directory**.
2. The path that is specified in the environment variables `LIBTC1V1_3 / LIBTC1V1_3_1 / LIBTC1V1_6`.
3. The default directory $\$(PRODDIR)\lib\[pcp1][pcp2]$.

Example

Suppose you call the linker as follows:

```
lpcp test.o --library-directory=c:\mylibs --library=c
```


First the linker looks in the directory `c:\mylibs` for library `libc.a` (this option). If it does not find the requested libraries, it looks in the directory that is set with the environment variables `LIBTC1V1_3` / `LIBTC1V1_3_1` / `LIBTC1V1_6`. Then the linker looks in the default directory `$(PRODDIR)\lib\[pcp1][pcp2]` for libraries.

Related information

[Linker option `--library`](#) (Link system library)

[Section 5.3.1, *How the Linker Searches Libraries*](#)

Linker option: **--link-only**

Menu entry

-

Command line syntax

`--link-only`

Description

With this option you suppress the locating phase. The linker stops after linking and informs you about unresolved references.

Related information

Control program option `--create=relocatable (-cl)` (Stop after linking)

Linker option: **--lsl-check**

Menu entry

-

Command line syntax

--lsl-check

Description

With this option the linker just checks the syntax of the LSL file(s) and exits. No linking or locating is performed. Use the option **--lsl-file** to specify the name of the Linker Script File you want to test.

Related information

Linker option **--lsl-file** (Linker script file)

Linker option **--lsl-dump** (Dump LSL info)

Section 5.7, *Controlling the Linker with a Script*

Linker option: **--lsl-dump**

Menu entry

-

Command line syntax

--lsl-dump[=*file*]

Description

With this option you tell the linker to dump the LSL part of the map file in a separate file, independent of the option **--map-file** (generate map file). If you do not specify a filename, the file `lpcp.ldf` is used.

Related information

Linker option **--map-file-format** (Map file formatting)

Linker option: `--lsl-file (-d)`

Menu entry

An LSL file can be generated when you create your TriCore project in Eclipse:

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

The New C/C++ Project wizard appears.

2. Fill in the project settings in each dialog and click **Next >** until the **TriCore Project Settings** appear.
3. Enable the option **Add Linker script file to the project** and click **Finish**.

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

The LSL file can be specified in the Properties dialog:

1. Select **Linker » Script File**.
2. Specify a LSL file in the **Linker script file (.lsl)** field.

Command line syntax

```
--lsl-file=file
```

```
-dfile
```

Description

A linker script file contains vital information about the core for the locating phase of the linker. A linker script file is coded in LSL and contains the following types of information:

- the architecture definition describes the core's hardware architecture.
- the memory definition describes the physical memory available in the system.
- the section layout definition describes how to locate sections in memory.

With this option you specify a linker script file to the linker. If you do not specify this option, the linker uses a default script file. You can specify the existing file `target.lsl` or the name of a manually written linker script file. You can use this option multiple times. The linker processes the LSL files in the order in which they appear on the command line.

Related information

[Linker option `--lsl-check`](#) (Check LSL file(s) and exit)

[Section 5.7, *Controlling the Linker with a Script*](#)

Linker option: --map-file (-M)

Menu entry

1. Select **Linker » Map File**.
2. Enable the option **Generate XML map file format (.mapxml) for map file viewer**.
3. (Optional) Enable the option **Generate map file**.
4. Enable or disable the types of information to be included.

Command line syntax

`--map-file[=file][:XML]`

`-M[file][:XML]`

Default (Eclipse): XML map file is generated

Default (linker): no map file is generated

Description

With this option you tell the linker to generate a linker map file. If you do not specify a filename and you specified the option `--output`, the linker uses the same basename as the output file with the extension `.map`. If you did not specify the option `--output`, the linker uses the file `task1.map`. Eclipse names the `.map` file after the project.

A linker map file is a text file that shows how the linker has mapped the sections and symbols from the various object files (`.o`) to the linked object file. A locate part shows the absolute position of each section. External symbols are listed per space with their absolute address, both sorted on symbol and sorted on address.

Related information

Linker option `--map-file-format` (Format map file)

Section 10.2, *Linker Map File Format*

Linker option: --map-file-format (-m)

Menu entry

1. Select **Linker » Map File**.
2. Enable the option **Generate XML map file format (.mapxml) for map file viewer**.
3. (Optional) Enable the option **Generate map file**.
4. Enable or disable the types of information to be included.

Command line syntax

`--map-file-format=flag,...`

`-mflags`

You can set the following flags:

+/-callgraph	c/C	Include call graph information
+/-removed	d/D	Include information on removed sections
+/-files	f/F	Include processed files information
+/-invocation	i/I	Include information on invocation and tools
+/-link	k/K	Include link result information
+/-locate	l/L	Include locate result information
+/-memory	m/M	Include memory usage information
+/-nonalloc	n/N	Include information of non-alloc sections
+/-overlay	o/O	Include overlay information
+/-statics	q/Q	Include module local symbols information
+/-crossref	r/R	Include cross references information
+/-lsl	s/S	Include processor and memory information
+/-rules	u/U	Include locate rules

Use the following options for predefined sets of flags:

<code>--map-file-format=0</code>	-m0	Link information Alias for -mCDfikLMNoQrSU
<code>--map-file-format=1</code>	-m1	Locate information Alias for -mCDfiKIMNoQRSU
<code>--map-file-format=2</code>	-m2	Most information Alias for -mcdfiklmNoQrSu

Default: `--map-file-format=2`

Description

With this option you specify which information you want to include in the map file.

On the command line you must use this option in combination with the option **--map-file (-M)**.

Related information

Linker option **--map-file** (Generate map file)

Section 10.2, *Linker Map File Format*

Linker option: `--misra-c-report`

Menu entry

-

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 *basename.mcr* is used.

Related information

C compiler option `--misrac` (MISRA-C checking)

Linker option: `--non-romable`

Menu entry

-

Command line syntax

`--non-romable`

Description

With this option you tell the linker that the application must not be located in ROM. The linker will locate all ROM sections, including a copy table if present, in RAM. When the application is started, the data sections are re-initialized and the BSS sections are cleared as usual.

This option is, for example, useful when you want to test the application in RAM before you put the final application in ROM. This saves you the time of flashing the application in ROM over and over again.

Related information

-

Linker option: `--no-rescan`

Menu entry

1. Select **Linker » Libraries**.
2. Disable the option **Rescan libraries to solve unresolved externals**.

Command line syntax

`--no-rescan`

Description

When the linker processes a library it searches for symbol definitions that are referenced by the objects and libraries processed so far. If the library contains a definition for an unresolved reference the linker extracts the object that contains the definition from the library. The linker processes object files and libraries in the order in which they appear on the command line.

When all objects and libraries are processed the linker checks if there are unresolved symbols left. If so, the default behavior of the linker is to rescan all libraries in the order given at the command line. The linker stops rescanning the libraries when all symbols are resolved, or when the linker could not resolve any symbol(s) during the rescan of all libraries. Notice that resolving one symbol may introduce new unresolved symbols.

With this option, you tell the linker to scan the object files and libraries only once. When the linker has not resolved all symbols after the first scan, it reports which symbols are still unresolved. This option is useful if you are building your own libraries. The libraries are most efficiently organized if the linker needs only one pass to resolve all symbols.

Related information

[Linker option `--first-library-first`](#) (Scan libraries in given order)

Linker option: **--no-rom-copy (-N)**

Menu entry

-

Command line syntax

`--no-rom-copy`

`-N`

Description

With this option the linker will not generate a ROM copy for data sections. A copy table is generated and contains entries to clear BSS sections. However, no entries to copy data sections from ROM to RAM are placed in the copy table.

The data sections are initialized when the application is downloaded. The data sections are not re-initialized when the application is restarted.

Related information

-

Linker option: `--no-warnings (-w)`

Menu entry

1. Select **Linker » Diagnostics**.

The Suppress warnings box shows the warnings that are currently suppressed.

2. To suppress a warning, click on the **Add** button in the **Suppress warnings** box.
3. Enter the numbers, separated by commas, of the warnings you want to suppress (for example 135, 136). Or you can use the **Add** button multiple times.
4. To suppress all warnings, enable the option **Suppress all warnings**.

Use the **Edit** and **Delete** button to change a warning number or to remove a number from the list.

Command line syntax

```
--no-warnings [=number, ...]
```

```
-w [number, ...]
```

Description

With this option you can suppresses all warning messages or specific warning messages.

On the command line this option works as follows:

- If you do not specify this option, all warnings are reported.
- If you specify this option but without numbers, all warnings are suppressed.
- If you specify this option with a number, only the specified warning is suppressed. You can specify the option `--no-warnings=number` multiple times.

Example

To suppress warnings 135 and 136, enter:

```
lpcp --no-warnings=135,136 test.o
```

Related information

[Linker option `--warnings-as-errors`](#) (Treat warnings as errors)

Linker option: --optimize (-O)

Menu entry

1. Select **Linker » Optimization**.
2. Select one or more of the following options:
 - Delete unreferenced sections
 - Use a 'first-fit decreasing' algorithm
 - Compress copy table
 - Delete duplicate code
 - Delete duplicate data

Command line syntax

`--optimize=flag,...`

`-Oflags`

You can set the following flags:

+/-delete-unreferenced-sections	c/C	Delete unreferenced sections from the output file
+/-first-fit-decreasing	I/L	Use a 'first-fit decreasing' algorithm to locate unrestricted sections in memory
+/-copytable-compression	t/T	Emit smart restrictions to reduce copy table size
+/-delete-duplicate-code	x/X	Delete duplicate code sections from the output file
+/-delete-duplicate-data	y/Y	Delete duplicate constant data from the output file

Use the following options for predefined sets of flags:

--optimize=0	-O0	No optimization Alias for -OCLTXY
--optimize=1	-O1	Default optimization Alias for -OcLtxy
--optimize=2	-O2	All optimizations Alias for -Ocltxy

Default: `--optimize=1`

Description

With this option you can control the level of optimization.

Related information

For details about each optimization see [Section 5.6, *Linker Optimizations*](#).

Linker option: --option-file (-f)

Menu entry

1. Select **Linker » Miscellaneous**.
2. Add the option **--option-file** to the **Additional options** field.

Be aware that the options in the option file are added to the linker options you have set in the other pages. Only in extraordinary cases you may want to use them in combination.

Command line syntax

`--option-file=file,...`

`-f file,...`

Description

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

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

Option files can also be generated on the fly, for example by the make utility. You can specify the option **--option-file** multiple times.

Format of an option file

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

```
--map-file=my.map           (generate a map file)
test.o                     (input file)
--library-directory=c:\mylibs (additional search path for system libraries)
```

Specify the option file to the linker:

```
lpcp --option-file=myoptions
```

This is equivalent to the following command line:

```
lpcp --map-file=my.map test.o --library-directory=c:\mylibs
```

Related information

-

Linker option: **--output (-o)**

Menu entry

1. Select **Linker » Output Format**.
2. Enable one or more output formats.

For some output formats you can specify a number of suboptions.

Eclipse always uses the project name as the basename for the output file.

Command line syntax

```
--output=[filename][:format[:addr_size][,space_name]]...
```

```
-o[filename][:format[:addr_size][,space_name]]...
```

You can specify the following formats:

ELF	ELF/DWARF
IHEX	Intel Hex
SREC	Motorola S-records

Description

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

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

You can use the **--output** option multiple times. This is useful to generate multiple output formats. With the first occurrence of the **--output** option you specify the basename (the filename without extension), which is used for subsequent **--output** options with no filename specified. If you do not specify a filename, or you do not specify the **--output** option at all, the linker uses the default basename `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. If they exist, any other address spaces are also emitted whereas their output files are named *filename_spacename* with the extension `.hex` or `.sre`.

If you do not specify *space_name*, or you specify a non-existing space, the default address space is filled in.

Use option **--chip-output (-c)** to create Intel Hex or Motorola S-record output files for each chip defined in the LSL file (suitable for loading into a PROM-programmer).

Example

To create the output file `myfile.hex` of the address space named `linear`, enter:

```
lpcp test.o --output=myfile.hex:IHEX:2,linear
```

If they exist, any other address spaces are emitted as well and are named `myfile_spacename.hex`.

Related information

[Linker option --chip-output](#) (Generate an output file for each chip)

[Linker option --hex-format](#) (Specify Hex file format settings)

Linker option: **--strip-debug (-S)**

Menu entry

1. Select **Linker » Miscellaneous**.
2. Enable the option **Strip symbolic debug information**.

Command line syntax

`--strip-debug`

`-S`

Description

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

Related information

-

Linker option: **--user-provided-initialization-code (-i)**

Menu entry

1. Select **Linker » Miscellaneous**.
2. Enable the option **Do not use standard copy table for initialization**.

Command line syntax

```
--user-provided-initialization-code
```

```
-i
```

Description

It is possible to use your own initialization code, for example, to save ROM space. With this option you tell the linker *not* to generate a copy table for initialize/clear sections. Use linker labels in your source code to access the positions of the sections when located.

If the linker detects references to the TASKING initialization code, an error is emitted: it is either the TASKING initialization routine or your own, not both.

Note that the options **--no-rom-copy** and **--non-romable**, may vary independently. The 'copytable-compression' optimization (**--optimize=t**) is automatically disabled when you enable this option.

Related information

[Linker option **--no-rom-copy**](#) (Do not generate ROM copy)

[Linker option **--non-romable**](#) (Application is not romable)

[Linker option **--optimize**](#) (Specify optimization)

Linker option: `--verbose (-v)` / `--extra-verbose (-vv)`

Menu entry

-

Command line syntax

`--verbose` / `--extra-verbose`

`-v` / `-vv`

Description

With this option you put the linker in *verbose* mode. The linker prints the link phases while it processes the files. In the *extra verbose* mode, the linker also prints the filenames and it shows which objects are extracted from libraries. With this option you can monitor the current status of the linker.

Related information

-

Linker option: --version (-V)

Menu entry

-

Command line syntax

`--version`

`-v`

Description

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

Example

```
lpcp --version
```

The linker does not link any files but displays the following version information:

```
TASKING VX-toolset for PCP: object linker   vx.yrz Build nnn  
Copyright 2006-year Altium BV             Serial# 00000000
```

Related information

-

Linker option: **--warnings-as-errors**

Menu entry

1. Select **Global Options**.
2. Enable the option **Treat warnings as errors**.

Command line syntax

--warnings-as-errors [=number, ...]

Description

When the linker detects an error or warning, it tries to continue the link process and reports other errors and warnings. When you use this option without arguments, you tell the linker to treat all warnings as errors. This means that the exit status of the linker will be non-zero after the detection of one or more linker warnings. As a consequence, the linker will not produce any output files.

You can also limit this option to specific warnings by specifying a comma-separated list of warning numbers.

Related information

Linker option **--no-warnings** (Suppress some or all warnings)

8.4. Control Program Options

The control program **ccpcp** facilitates the invocation of the various components of the PCP toolset from a single command line.

Options in Eclipse versus options on the command line

Eclipse invokes the compiler, assembler and linker via the control program. Therefore, it uses the syntax of the control program to pass options and files to the tools. The control program processes command line options either by itself, or, when the option is unknown to the control program, it looks whether it can pass the option to one of the other tools. However, for directly passing an option to the compiler, assembler or linker, it is recommended to use the control program options **--pass-c**, **--pass-assembler**, **--pass-linker**.

See the previous sections for details on the options of the tools.

Short and long option names

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

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

```
ccpcp -Wc-Oac test.c
ccpcp --pass-c=--optimize=+coalesce,+cse test.c
```

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

Control program option: `--address-size`

Menu entry

1. Select **Linker » Output Format**.
2. Enable the option **Generate Intel Hex format file** and/or **Generate S-records file**.
3. Specify the **Size of addresses**.

Eclipse always uses the project name as the basename for the output file.

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 to be emitted in the output files.

With this option you can specify the size of the addresses in bytes (record length). For Intel Hex you can use the values **1**, **2** or **4** bytes (default). For Motorola-S you can specify: **2** (S1 records), **3** (S2 records) or **4** bytes (S3 records, default).

If you do not specify `addr_size`, the default address size is generated.

Example

To create the SREC file `test.sre` with S1 records, type:

```
ccpcp --format=SREC --address-size=2 test.c
```

Related information

[Control program option `--format`](#) (Set linker output format)

[Control program option `--output`](#) (Output file)

Control program option: `--case-insensitive`

Menu entry

1. Select **Assembler » Symbols**.
2. Enable the option **Case insensitive identifiers**.

Command line syntax

`--case-insensitive`

Default: case sensitive

Description

With this option you tell the assembler not to distinguish between upper and lower case characters. By default the assembler considers upper and lower case characters as different characters.

Assembly source files that are generated by the compiler must always be assembled case sensitive. When you are writing your own assembly code, you may want to specify the case insensitive mode.

Example

When assembling case insensitive, the label `LabelName` is the same label as `labelname`.

```
ccpcp --case-insensitive test.src
```

Related information

[Assembler option `--case-insensitive`](#)

[Assembler control `\$CASE`](#)

Control program option: **--check**

Menu entry

-

Command line syntax

--check

Description

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

The compiler/assembler reports any warnings and/or errors.

This option is available on the command line only.

Related information

C compiler option **--check** (Check syntax)

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

Control program option: `--cpu (-C)`

Menu entry

1. Expand **C/C++ Build** and select **Processor**.
2. From the **Processor Selection** list, select a processor or select **User defined TriCore ...**

Command line syntax

```
--cpu=cpu
```

```
-Ccpu
```

Description

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

Based on this option the compiler always includes the special function register file `regcpu.sfr`, and the assembler includes the file `regcpu.def`, unless you specify option `--no-tasking-sfr`.

Example

To generate the file `test.elf` for the TC1165 processor, enter:

```
ccpcp --cpu=tc1165 test.c
```

Related information

Control program option `--no-tasking-sfr` (Do not include SFR file)

Section 1.2.4, *Accessing Hardware from C*

Control program option: `--create (-c)`

Menu entry

-

Command line syntax

`--create[=stage]`

`-c[stage]`

You can specify the following stages:

relocatable	l	Stop after the files are linked to a linker object file (<code>.out</code>)
mil	m	Stop after C files are compiled to MIL (<code>.mil</code>)
object	o	Stop after the files are assembled to objects (<code>.o</code>)
assembly	s	Stop after C files are compiled to assembly (<code>.src</code>)

Default (without flags): `--create=object`

Description

Normally the control program generates an absolute object file of the specified output format from the file you supplied as input. With this option you tell the control program to stop after a certain number of phases.

Example

To generate the object file `test.o`:

```
ccpcp --create test.c
```

The control program stops after the file is assembled. It does not link nor locate the generated output.

Related information

[Linker option `--link-only`](#) (Link only, no locating)

Control program option: `--debug-info (-g)`

Menu entry

1. Select **C Compiler » Debugging**.
2. To **generate symbolic debug information**, select **Default**, **Small set** or **Full**.
To disable the generation of debug information, select **None**.

Command line syntax

`--debug-info`

`-g`

Description

With this option you tell the control program to include debug information in the generated object file.

The control program passes the option `--debug-info (-g)` to the C compiler and calls the assembler with `--debug-info=+smart,+local (-gsi)`.

Related information

[C compiler option `--debug-info`](#) (Generate symbolic debug information)

[Assembler option `--debug-info`](#) (Generate symbolic debug information)

Control program option: `--define (-D)`

Menu entry

1. Select **C Compiler » Preprocessing** and/or **Assembler » Preprocessing**.

The Defined symbols box right-below shows the symbols that are currently defined.

2. To define a new symbol, click on the **Add** button in the **Defined symbols** box.
3. Type the symbol definition (for example, `demo=1`)

Use the **Edit** and **Delete** button to change a macro definition or to remove a macro from the list.

Command line syntax

```
--define=macro_name[=macro_definition]
```

```
-Dmacro_name[=macro_definition]
```

Description

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

You can specify as many macros as you like. Simply use the **Add** button to add new macro definitions.

On the command line, use the option `--define (-D)` multiple times. If the command line exceeds the limit of the operating system, you can define the macros in an option file which you then must specify to the compiler with the option `--option-file (-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.

The control program passes the option `--define (-D)` to the compiler and the assembler.

Example

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

```
void main( void )
{
  #if DEMO
    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:


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

Note that both invocations have the same effect.

The next example shows how to define a macro with arguments. Note that the macro name and definition are placed between double quotes because otherwise the spaces would indicate a new option.

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

Related information

Control program option **--undefine** (Remove preprocessor macro)

Control program option **--option-file** (Specify an option file)

Control program option: **--dep-file**

Menu entry

-

Command line syntax

--dep-file[=*file*]

Description

With this option you tell the compiler to generate dependency lines that can be used in a Makefile. In contrast to the option **--preprocess=+make**, the dependency information will be generated in addition to the normal output file.

By default, the information is written to a file with extension `.d` (one for every input file). When you specify a filename, all dependencies will be combined in the specified file.

Example

```
ccpcp --dep-file=test.dep -t test.c
```

The compiler compiles the file `test.c`, which results in the output file `test.src`, and generates dependency lines in the file `test.dep`.

Related information

Control program option **--preprocess=+make** (Generate dependencies for make)

Control program option: --diag

Menu entry

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

The Problems view is added to the current perspective.

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

A popup menu appears.

3. Select **Detailed Diagnostics Info**.

A dialog box appears with additional information.

Command line syntax

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

You can set the following output formats:

html	HTML output.
rtf	Rich Text Format.
text	ASCII text.

Default format: text

Description

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

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

Example

To display an explanation of message number 103, enter:

```
ccpcp --diag=103
```

This results in message 103 with explanation.

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

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

Related information

[Section 3.8, *C Compiler Error Messages*](#)

Control program option: `--dry-run (-n)`

Menu entry

-

Command line syntax

`--dry-run`

`-n`

Description

With this option you put the control program in verbose mode. The control program prints the invocations of the tools it would use to process the files without actually performing the steps.

Related information

Control program option `--verbose` (Verbose output)

Control program option: **--error-file**

Menu entry

-

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.

Example

To write errors to error files instead of stderr, enter:

```
ccpcp --error-file test.c
```

Related information

Control Program option **--warnings-as-errors** (Treat warnings as errors)

Control program option: `--format`

Menu entry

1. Select **Linker » Output Format**.
2. Enable the option **Generate Intel Hex format file** and/or **Generate S-records file**.
3. Optionally, specify the **Size of addresses**.

Eclipse always uses the project name as the basename for the output file.

Command line syntax

`--format=format`

You can specify the following formats:

ELF	ELF/DWARF
IHEX	Intel Hex
SREC	Motorola S-records

Description

With this option you specify the output format for the resulting (absolute) object file. The default output format is ELF/DWARF, which can directly be used by the debugger.

If you choose IHEX or SREC, you can additionally specify the address size of the chosen format (option `--address-size`).

Example

To generate a Motorola S-record output file:

```
ccpcp --format=SREC test1.c test2.c --output=test.sre
```

Related information

[Control program option `--address-size`](#) (Set address size for linker IHEX/SREC files)

[Control program option `--output`](#) (Output file)

[Linker option `--chip-output`](#) (Generate an output file for each chip)

Control program option: `--fp-trap`

Menu entry

1. Select **Linker » Libraries**.
2. Enable the option **Use trapped floating-point library**.

Command line syntax

`--fp-trap`

Description

By default the control program uses the non-trapping floating-point library (`libfp.a`). With this option you tell the control program to use the trapping floating-point library (`libfpt.a`).

If you use the trapping floating-point library, exceptional floating-point cases are intercepted and can be handled separately by an application defined exception handler. Using this library decreases the execution speed of your application.

Related information

[Section 5.3, *Linking with Libraries*](#)

Control program option: --help (-?)

Menu entry

-

Command line syntax

```
--help[=item]
```

```
-?
```

You can specify the following argument:

options	Show extended option descriptions
----------------	-----------------------------------

Description

Displays an overview of all command line options. When you specify the argument **options** you can list detailed option descriptions.

Example

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

```
ccpcp -?  
ccpcp --help  
ccpcp
```

To see a detailed description of the available options, enter:

```
ccpcp --help=options
```

Related information

-

Control program option: **--include-directory (-I)**

Menu entry

1. Select **C Compiler » Include Paths**.

The Include paths box shows the directories that are added to the search path for include files.

2. To define a new directory for the search path, click on the **Add** button in the **Include paths** box.
3. Type or select a path.

Use the **Edit** and **Delete** button to change a path or to remove a path from the list.

Command line syntax

`--include-directory=path,...`

`-Ipath,...`

Description

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

The control program passes this option to the compiler and the assembler.

Example

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

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

You can call the control program as follows:

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

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

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

Related information

[C compiler option **--include-directory**](#) (Add directory to include file search path)

[C compiler option **--include-file**](#) (Include file at the start of a compilation)

Control program option: `--iso`

Menu entry

1. Select **C Compiler » Language**.
2. From the **Comply to C standard** list, select **ISO C99** or **ISO C90**.

Command line syntax

```
--iso={90 | 99}
```

Default: `--iso=99`

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.

Independent of the chosen ISO standard, the control program always links libraries with C99 support.

Example

To select the ISO C90 standard on the command line:

```
ccpcp --iso=90 test.c
```

Related information

[C compiler option `--iso` \(ISO C standard\)](#)

Control program option: **--keep-output-files (-k)**

Menu entry

Eclipse *always* removes generated output files when an error occurs.

Command line syntax

--keep-output-files

-k

Description

If an error occurs during the 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.

The control program passes this option to the compiler, assembler and linker.

Example

```
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

[C compiler option **--keep-output-files**](#)

[Assembler option **--keep-output-files**](#)

[Linker option **--keep-output-files**](#)

Control program option: `--keep-temporary-files (-t)`

Menu entry

1. Select **Global Options**.
2. Enable the option **Keep temporary files**.

Command line syntax

```
--keep-temporary-files
```

```
-t
```

Description

By default, the control program removes intermediate files like the `.src` file (result of the compiler phase) and the `.o` file (result of the assembler phase).

With this option you tell the control program to keep temporary files it generates during the creation of the absolute object file.

Example

```
ccpcp --keep-temporary-files test.c
```

The control program keeps all intermediate files it generates while creating the absolute object file `test.elf`.

Related information

-

Control program option: `--library (-l)`

Menu entry

1. Select **Linker » Libraries**.

The Libraries box shows the list of libraries that are linked with the project.

2. To add a library, click on the **Add** button in the **Libraries** box.
3. Type or select a library (including its path).
4. Optionally, disable the option **Link default libraries**.

Use the **Edit** and **Delete** button to change a library name or to remove a library from the list.

Command line syntax

`--library=name`

`-lname`

Description

With this option you tell the linker via the control program to use system library `libname.a`, where `name` is a string. The linker first searches for system libraries in any directories specified with `--library-directory`, then in the directories specified with the environment variables `LIBTCLV1_3` / `LIBTCLV1_3_1` / `LIBTCLV1_6`, unless you used the option `--ignore-default-library-path`.

Example

To search in the system library `libc.a` (C library):

```
ccpcp test.o mylib.a --library=c
```

The linker links the file `test.o` and first looks in library `mylib.a` (in the current directory only), then in the system library `libc.a` to resolve unresolved symbols.

Related information

Control program option `--no-default-libraries` (Do not link default libraries)

Control program option `--library-directory` (Additional search path for system libraries)

Section 5.3, *Linking with Libraries*

Control program option: **--library-directory (-L) / --ignore-default-library-path**

Menu entry

1. Select **Linker » Libraries**.

The Library search path box shows the directories that are added to the search path for library files.

2. To define a new directory for the search path, click on the **Add** button in the **Library search path** box.
3. Type or select a path.

Use the **Edit** and **Delete** button to change a path or to remove a path from the list.

Command line syntax

```
--library-directory=path,...
```

```
-Lpath,...
```

```
--ignore-default-library-path
```

```
-L
```

Description

With this option you can specify the path(s) where your system libraries, specified with the option **--library (-l)**, are located. If you want to specify multiple paths, use the option **--library-directory** for each separate path.

The default path is $\$(PRODDIR)\lib\[pcp1][pcp2]$.

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_3` / `LIBTC1V1_3_1` / `LIBTC1V1_6`. So, the linker ignores steps 2 and 3 as listed below.

The priority order in which the linker searches for system libraries specified with the option **--library (-l)** is:

1. The path that is specified with the option **--library-directory**.
2. The path that is specified in the environment variables `LIBTC1V1_3` / `LIBTC1V1_3_1` / `LIBTC1V1_6`.
3. The default directory $\$(PRODDIR)\lib\[pcp1][pcp2]$.

Example

Suppose you call the control program as follows:

TASKING VX-toolset for PCP User Guide

```
ccpcp test.c --library-directory=c:\mylibs --library=c
```

First the linker looks in the directory `c:\mylibs` for library `libc.a` (this option). If it does not find the requested libraries, it looks in the directory that is set with the environment variables `LIBTC1V1_3` / `LIBTC1V1_3_1` / `LIBTC1V1_6`. Then the linker looks in the default directory `$(PRODDIR)\lib\[pcp1][pcp2]` for libraries.

Related information

Control program option **--library** (Link system library)

Section 5.3.1, *How the Linker Searches Libraries*

Control program option: `--list-files`

Menu entry

-

Command line syntax

```
--list-files[=file]
```

Default: no list files are generated

Description

With this option you tell the assembler via the control program to generate a list file for each specified input file. A list file shows the generated object code and the relative addresses. Note that the assembler generates a relocatable object file with relative addresses.

With the optional *file* you can specify a name for the list file. This is only possible if you specify only one input file to the control program. If you do not specify a file name, or you specify more than one input file, the control program names the generated list file(s) after the specified input file(s) with extension `.lst`.

Note that object files and library files are not counted as input files.

Related information

[Assembler option `--list-file`](#) (Generate list file)

[Assembler option `--list-format`](#) (Format list file)

Control program option: **--lsl-file (-d)**

Menu entry

An LSL file can be generated when you create your TriCore project in Eclipse:

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

The New C/C++ Project wizard appears.

2. Fill in the project settings in each dialog and click **Next >** until the **TriCore Project Settings** appear.
3. Enable the option **Add Linker script file to the project** and click **Finish**.

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

The LSL file can be specified in the Properties dialog:

1. Select **Linker » Script File**.
2. Specify a LSL file in the **Linker script file (.lsl)** field.

Command line syntax

```
--lsl-file=file,...
```

```
-dfile,...
```

Description

A linker script file contains vital information about the core for the locating phase of the linker. A linker script file is coded in LSL and contains the following types of information:

- the architecture definition describes the core's hardware architecture.
- the memory definition describes the physical memory available in the system.
- the section layout definition describes how to locate sections in memory.

With this option you specify a linker script file via the control program to the linker. If you do not specify this option, the linker uses a default script file. You can specify the existing file *target.lsl* or the name of a manually written linker script file. You can use this option multiple times. The linker processes the LSL files in the order in which they appear on the command line.

Related information

[Section 5.7, Controlling the Linker with a Script](#)

Control program option: `--make-target`

Menu entry

-

Command line syntax

`--make-target=name`

Description

With this option you can overrule the default target name in the make dependencies generated by the options `--preprocess=+make` (`-Em`) and `--dep-file`. The default target name is the basename of the input file, with extension `.o`.

Example

```
ccpcp --preprocess=+make --make-target=../mytarget.o test.c
```

The compiler generates dependency lines with the default target name `../mytarget.o` instead of `test.o`.

Related information

Control program option `--preprocess=+make` (Generate dependencies for make)

Control program option `--dep-file` (Generate dependencies in a file)

Control program option: `--mil-link` / `--mil-split`

Menu entry

1. Select **C Compiler » Optimization**.
2. Enable the option **Build for application wide optimizations (MIL linking)**.

Command line syntax

```
--mil-link  
--mil-split[=file,...]
```

Description

With option `--mil-link` the C compiler links the optimized intermediate representation (MIL) of all input files and MIL libraries specified on the command line in the compiler. The result is one single module that is optimized another time.

Option `--mil-split` does the same as option `--mil-link`, but in addition, the resulting MIL representation is written to a file with the suffix `.mil` and the C compiler also splits the MIL representation and writes it to separate files with suffix `.ms`. One file is written for each input file or MIL library specified on the command line. The `.ms` files are only updated on a change.

With option `--mil-split` you can perform application-wide optimizations during the frontend phase by specifying all modules at once, and still invoke the backend phase one module at a time to reduce the total compilation time.

Optionally, you can specify another filename for the `.ms` file the C compiler generates. Without an argument, the basename of the C source file is used to create the `.ms` filename. Note that if you specify a filename, you have to specify one filename for every input file.

Related information

[Section 3.1, *Compilation Process*](#)

[C compiler option `--mil` / `--mil-split`](#)

Control program option: `--no-default-libraries`

Menu entry

1. Select **Linker » Libraries**.
2. Disable the option **Link default libraries**.

Command line syntax

```
--no-default-libraries
```

Description

By default the control program specifies the standard C libraries (C99) and run-time library to the linker. With this option you tell the control program not to specify the standard C libraries and run-time library to the linker.

In this case you must specify the libraries you want to link to the linker with the option `--library=library_name` or pass the libraries as files on the command line. The control program recognizes the option `--library (-l)` as an option for the linker and passes it as such.

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 (`libc.a`) and avoid unresolved externals:

```
ccpcp --no-default-libraries --library=c test.c
```

Related information

[Control program option `--library`](#) (Link system library)

[Section 5.3.1, *How the Linker Searches Libraries*](#)

Control program option: **--no-map-file**

Menu entry

1. Select **Linker » Map File**.
2. Disable the option **Generate map file**.

Command line syntax

--no-map-file

Description

By default the control program tells the linker to generate a linker map file.

A linker map file is a text file that shows how the linker has mapped the sections and symbols from the various object files (.o) to the linked object file. A locate part shows the absolute position of each section. External symbols are listed per space with their absolute address, both sorted on symbol and sorted on address.

With this option you prevent the generation of a map file.

Related information

-

Control program option: `--no-tasking-sfr`

Menu entry

1. Select **C Compiler** » **Preprocessing**.
2. Disable the option **Automatic inclusion of '.sfr' file**.
3. Select **Assembler** » **Preprocessing**.
4. Disable the option **Automatic inclusion of '.def' file**.

Command line syntax

`--no-tasking-sfr`

Description

Normally, the C compiler and assembler includes a special function register (SFR) file before compiling/assembling. The compiler and assembler automatically select the SFR file belonging to the target you selected on the **Processor** page (control program option `--cpu`).

With this option the compiler and assembler do 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.

Related information

[Control program option `--cpu`](#) (Select processor)

[Section 1.2.4, *Accessing Hardware from C*](#)

Control program option: `--no-warnings (-w)`

Menu entry

1. Select **C Compiler » Diagnostics**.

The Suppress C compiler warnings box shows the warnings that are currently suppressed.

2. To suppress a warning, click on the **Add** button in the **Suppress warnings** box.
3. Enter the numbers, separated by commas or as a range, of the warnings you want to suppress (for example 537, 538). Or you can use the **Add** button multiple times.
4. To suppress all warnings, enable the option **Suppress all warnings**.

Use the **Edit** and **Delete** button to change a warning number or to remove a number from the list.

Command line syntax

```
--no-warnings[=number[-number],...]
```

```
-w[number[-number],...]
```

Description

With this option you can suppresses all warning messages for the various tools or specific control program warning messages.

On the command line this option works as follows:

- If you do not specify this option, all warnings are reported.
- If you specify this option but without numbers, all warnings of all tools are suppressed.
- If you specify this option with a number or a range, only the specified control program warnings are suppressed. You can specify the option `--no-warnings=number` multiple times.

Example

To suppress all warnings for all tools, enter:

```
ccpcp test.c --no-warnings
```

Related information

Control program option `--warnings-as-errors` (Treat warnings as errors)

Control program option: `--option-file (-f)`

Menu entry

-

Command line syntax

```
--option-file=file,...
```

```
-f file,...
```

Description

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

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

You can specify the option `--option-file` multiple times.

Format of an option file

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

TASKING VX-toolset for PCP User Guide

```
--debug-info  
--define=DEMO=1  
test.c
```

Specify the option file to the control program:

```
ccpcp --option-file=myoptions
```

This is equivalent to the following command line:

```
ccpcp --debug-info --define=DEMO=1 test.c
```

Related information

-

Control program option: **--output (-o)**

Menu entry

Eclipse always uses the project name as the basename for the output file.

Command line syntax

```
--output=file
```

```
-o file
```

Description

By default, the control program generates a file with the same basename as the first specified input file. With this option you specify another name for the resulting absolute object file.

The default output format is ELF/DWARF, but you can specify another output format with option **--format**.

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 --output=result.elf test.c prog.c
```

Related information

Control program option **--format** (Set linker output format)

Linker option **--output** (Output file)

Linker option **--chip-output** (Generate an output file for each chip)

Control program option: **--pass (-W)**

Menu entry

1. Select **C Compiler » Miscellaneous** or **Assembler » Miscellaneous** or **Linker » Miscellaneous**.
2. Add an option to the **Additional options** field.

*Be aware that the options in the option file are added to the options you have set in the other pages. Only in extraordinary cases you may want to use them in combination. The assembler options are preceded by **-Wa** and the linker options are preceded by **-Wl**. For the C options you have to do this manually.*

Command line syntax

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

Description

With this option you tell the control program to call a tool with the specified option. The control program does not use or interpret the option itself, but specifies it directly to the tool which it calls.

Example

To pass the option **--verbose** directly to the linker, enter:

```
ccpcp --pass-linker=--verbose test.c
```

Related information

-

Control program option: `--preprocess (-E) / --no-preprocessing-only`

Menu entry

1. Select **C Compiler » Preprocessing**.
2. Enable the option **Store preprocessor output in <file>.pre**.
3. (Optional) Enable the option **Keep comments in preprocessor output**.
4. (Optional) Enable the option **Keep #line info in preprocessor output**.

Command line syntax

`--preprocess [=flags]`

`-E[flags]`

`--no-preprocessing-only`

You can set the following flags:

+/-comments	c/C	keep comments
+/-includes	i/I	generate a list of included source files
+/-list	l/L	generate a list of macro definitions
+/-make	m/M	generate dependencies for make
+/-noline	p/P	strip #line source position information

Default: `-ECILMP`

Description

With this option you tell the compiler to preprocess the C source. The C compiler sends the preprocessed output to the file `name.pre` (where `name` is the name of the C source file to compile). Eclipse also compiles the C source.

On the command line, the control program stops after preprocessing. If you also want to compile the C source you can specify the option `--no-preprocessing-only`. In this case the control program calls the compiler twice, once with option `--preprocess` and once for a regular compilation.

With `--preprocess=+comments` you tell the preprocessor to keep the comments from the C source file in the preprocessed output.

With `--preprocess=+includes` the compiler will generate a list of all included source files. The preprocessor output is discarded.

With `--preprocess=+list` the compiler will generate a list of all macro definitions. The preprocessor output is discarded.

TASKING VX-toolset for PCP User Guide

With **--preprocess=+make** the compiler will generate dependency lines that can be used in a Makefile. The information is written to a file with extension `.d`. The preprocessor output is discarded. The default target name is the basename of the input file, with the extension `.o`. With the option **--make-target** you can specify a target name which overrules the default target name.

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

Example

```
ccpcp --preprocess=+comments,-make,-noline --no-preprocessing-only test.c
```

The compiler preprocesses the file `test.c` and sends the output to the file `test.pre`. Comments are included but no dependencies are generated and the line source position information is not stripped from the output file. Next, the control program calls the compiler, assembler and linker to create the final object file `test.elf`

Related information

Control program option **--dep-file** (Generate dependencies in a file)

Control program option **--make-target** (Specify target name for **-Em** output)

Control program option: `--silicon-bug`

Menu entry

1. Expand **C/C++ Build** and select **Processor**.
2. From the **Processor Selection** list, select a processor.

*The **CPU Problem Bypasses and Checks** box shows the available workarounds/checks available for the selected processor.*

3. (Optional) Select **Show all CPU problem bypasses and checks**.
4. Click **Select All** or select one or more individual options.

Command line syntax

```
--silicon-bug=arg,...
```

For a list of available arguments refer to the description of option `--silicon-bug` of the compiler and assembler. Depending on the available arguments this option is passed to the compiler and/or assembler.

Description

With this option the control program tells the compiler/assembler to use software workarounds for some CPU functional problems. Please refer to [Chapter 13, CPU Problem Bypasses and Checks](#) for more information about the individual problems and workarounds.

Example

To enable workarounds for problem PCP_TC.038, enter:

```
cctc --silicon-bug=pcp-tc038 test.c
```

Related information

[Chapter 13, CPU Problem Bypasses and Checks](#)

C compiler option `--silicon-bug`

Assembler option `--silicon-bug`

Control program option: `--static`

Menu entry

-

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

-

Control program option: --uchar (-u)

Menu entry

1. Select **C Compiler » Language**.
2. Enable the option **Treat "char" variables as unsigned**.

Command line syntax

`--uchar`

`-u`

Description

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

Related information

Section 1.1, *Data Types*

Control program option: **--undefine (-U)**

Menu entry

1. Select **C Compiler » Preprocessing**

The Defined symbols box shows the symbols that are currently defined.

2. To remove a defined symbol, select the symbol in the **Defined symbols** box and click on the **Delete** button.

Command line syntax

--undefine=*macro_name*

-U*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:

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

The control program passes the option **--undefine (-U)** to the compiler.

Example

To undefine the predefined macro `__TASKING__`:

```
ccpcp --undefine=__TASKING__ test.c
```

Related information

Control program option **--define** (Define preprocessor macro)

Section 1.6, *Predefined Preprocessor Macros*

Control program option: **--verbose (-v)**

Menu entry

1. Select **Global Options**.
2. Enable the option **Verbose mode of control program**.

Command line syntax

--verbose

-v

Description

With this option you put the control program in verbose mode. The control program performs its tasks while it prints the steps it performs to stdout.

Related information

Control program option **--dry-run** (Verbose output and suppress execution)

Control program option: `--version (-V)`

Menu entry

-

Command line syntax

`--version`

`-V`

Description

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

Related information

-

Control program option: **--warnings-as-errors**

Menu entry

1. Select **Global Options**.
2. Enable the option **Treat warnings as errors**.

Command line syntax

```
--warnings-as-errors[=number[-number], . . .]
```

Description

If one of the tools encounters an error, it stops processing the file(s). With this option you tell the tools to treat warnings as errors or treat specific control program warning messages as errors:

- If you specify this option but without numbers, all warnings are treated as errors.
- If you specify this option with a number or a range, only the specified control program warnings are treated as an error. You can specify the option **--warnings-as-errors=*number*** multiple times.

Use one of the **--pass-*tool*** options to pass this option directly to a tool when a specific warning for that tool must be treated as an error. For example, use **--pass-c=**--warnings-as-errors**=*number*** to treat a specific C compiler warning as an error.

Related information

Control program option **--no-warnings** (Suppress some or all warnings)

Control program option **--pass** (Pass option to tool)

8.5. Make Utility Options

When you build a project in Eclipse, Eclipse generates a makefile and uses the make utility **mkpcp** to build all your files. However, you can also use the make utility directly from the command line to build your project.

The invocation syntax is:

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

This section describes all options for the make utility. The make utility is a command line tool so there are no equivalent options in Eclipse.

For detailed information about the make utility and using makefiles see [Section 6.2, Make Utility mkpcp](#).

Defining Macros

Command line syntax

```
macro_name[=macro_definition]
```

Description

With this argument you can define a macro and specify it to the make utility.

A macro definition remains in existence during the execution of the makefile, even when the makefile recursively calls the make utility again. In the recursive call, the macro acts as an environment variable. This means that it is overruled by definitions in the recursive call. Use the option **-e** to prevent this.

You can specify as many macros as you like. If the command line exceeds the limit of the operating system, you can define the macros in an option file which you then must specify to the make utility with the option **-m** *file*.

Defining macros on the command line is, for example, useful in combination with conditional processing as shown in the example below.

Example

Consider the following makefile with conditional rules to build a demo program and a real program:

```
ifdef DEMO          # the value of DEMO is of no importance
    real.elf : demo.o main.o
                lpcp demo.o main.o -lc -lfp
else
    real.elf : real.o main.o
                lpcp real.o main.o -lc -lfp
endif
```

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

```
mkpcp real.elf DEMO=1
```

In both cases the absolute object file `real.elf` is created but depending on the DEMO flag it is linked with `demo.o` or with `real.o`.

Related information

Make utility option **-e** (Environment variables override macro definitions)

Make utility option **-m** (Name of invocation file)

Make utility option: -?

Command line syntax

-?

Description

Displays an overview of all command line options.

Example

The following invocation displays a list of the available command line options:

```
mkpcp -?
```

Related information

-

Make utility option: -a

Command line syntax

`-a`

Description

Normally the make utility rebuilds only those files that are out of date. With this option you tell the make utility to rebuild *all* files, without checking whether they are out of date.

Example

```
mkpcp -a
```

Rebuilds all your files, regardless of whether they are out of date or not.

Related information

-

Make utility option: **-c**

Command line syntax

-c

Description

Eclipse uses this option when you create sub-projects. In this case the make utility calls another instance of the make utility for the sub-project. With the option **-c**, the make utility runs as a child process of the current make.

The option **-c** overrides the option **-err**.

Example

```
mkpcp -c
```

The make utility runs its commands as a child processes.

Related information

[Make utility option **-err**](#) (Redirect error message to file)

Make utility option: -D / -DD

Command line syntax

-D
-DD

Description

With the option **-D** the make utility prints every line of the makefile to standard output as it is read by **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

-

Make utility option: **-d/ -dd**

Command line syntax

-d
-dd

Description

With the option **-d** the make utility shows which files are out of date and thus need to be rebuild. The option **-dd** gives more detail than the option **-d**.

Example

```
mkpcp -d
```

Shows which files are out of date and rebuilds them.

Related information

-

Make utility option: -e

Command line syntax

`-e`

Description

If you use macro definitions, they may overrule the settings of the environment variables. With the option `-e`, the settings of the environment variables are used even if macros define otherwise.

Example

```
mkpcp -e
```

The make utility uses the settings of the environment variables regardless of macro definitions.

Related information

-

Make utility option: **-err**

Command line syntax

`-err file`

Description

With this option the make utility redirects error messages and verbose messages to a specified file.

With the option **-s** the make utility only displays error messages.

Example

```
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)

[Make utility option **-c**](#) (Run as child process)

Make utility option: -f

Command line syntax

`-f my_makefile`

Description

By default the make utility uses the file `makefile` to build your files.

With this option you tell the make utility to use the specified file instead of the file `makefile`. Multiple `-f` options act as if all the makefiles were concatenated in a left-to-right order.

If you use `'-'` instead of a makefile name it means that the information is read from `stdin`.

Example

```
mkpcp -f mymake
```

The make utility uses the file `mymake` to build your files.

Related information

-

Make utility option: **-G**

Command line syntax

-G *path*

Description

Normally you must call the make utility from the directory where your makefile and other files are stored.

With the option **-G** you can call the make utility from within another directory. The *path* is the path to the directory where your makefile and other files are stored and can be absolute or relative to your current directory.

Example

Suppose your makefile and other files are stored in the directory `..\myfiles`. You can call the make utility, for example, as follows:

```
mkpcp -G ..\myfiles
```

Related information

-

Make utility option: -i

Command line syntax

`-i`

Description

When an error occurs during the make process, the make utility exits with a certain exit code.

With the option `-i`, the make utility exits without an error code, even when errors occurred.

Example

```
mkpcp -i
```

The make utility exits without an error code, even when an error occurs.

Related information

-

Make utility option: -K

Command line syntax

-K

Description

With this option the make utility keeps temporary files it creates during the make process. The make utility stores temporary files in the directory that you have specified with the environment variable TMPDIR or in the default 'temp' directory of your system when the TMPDIR environment variable is not specified.

Example

```
mkpcp -K
```

The make utility preserves all temporary files.

Related information

-

Make utility option: -k

Command line syntax

-k

Description

When during the make process the make utility encounters an error, it stops rebuilding your files.

With the option **-k**, the make utility only stops building the target that produced the error. All other targets defined in the makefile are built.

Example

```
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**)

Make utility option: -m

Command line syntax

`-m file`

Description

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

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

You can specify the option `-m` multiple times.

If you use '-' instead of a filename it means that the options are read from `stdin`.

Format of an option file

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

```
-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

-

Make utility option: -n

Command line syntax

-n

Description

With this option you tell the make utility to perform a dry run. The make utility shows what it would do but does not actually perform these tasks.

This option is for example useful to quickly inspect what would happen if you call the make utility.

Example

```
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)

Make utility option: -p

Command line syntax

-p

Description

Normally, if a command in a target rule in a makefile returns an error or when the target construction is interrupted, the make utility removes that target file. With this option you tell the make utility to make all target files precious. This means that all dependency files are never removed.

Example

```
mkpcp -p
```

The make utility never removes target dependency files.

Related information

Special target `.PRECIOUS` in [Section 6.2.2.1, *Targets and Dependencies*](#)

Make utility option: -q

Command line syntax

`-q`

Description

With this option the make utility does not perform any tasks but only returns an exit code. A zero status indicates that all target files are up to date, a non-zero status indicates that some or all target files are out of date.

Example

```
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

-

Make utility option: -r

Command line syntax

`-r`

Description

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

With this option you tell the make utility not to read `mkpcp.mk` and to rely fully on the make rules in the makefile.

Example

```
mkpcp -r
```

The make utility does not read the implicit make rules in `mkpcp.mk`.

Related information

-

Make utility option: **-S**

Command line syntax

-S

Description

With this option you cancel the effect of the option **-k**. This is only necessary in a recursive make where the option **-k** might be inherited from the top-level make via MAKEFLAGS or if you set the option **-k** in the environment variable MAKEFLAGS.

With this option you tell the make utility not to read `mkpcp.mk` and to rely fully on the make rules in the makefile.

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)

Make utility option: -s

Command line syntax

-s

Description

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

Example

```
mkpcp -s
```

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

Related information

[Make utility option -n](#) (Perform a dry run)

Make utility option: -t

Command line syntax

`-t`

Description

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

Example

```
mkpcp -t
```

The make utility updates out-of-date files by giving them a new date and time stamp. The files are not actually rebuild.

Related information

-

Make utility option: -time

Command line syntax

`-time`

Description

With this option you tell the make utility to display the current date and time on standard output.

Example

```
mkpcp -time
```

The make utility displays the current date and time and updates out-of-date files.

Related information

-

Make utility option: -V

Command line syntax

-V

Description

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

Example

```
mkpcp -V
```

The make utility displays the version information but does not perform any tasks.

```
TASKING VX-toolset for PCP: program builder   vx.yrz Build nnn  
Copyright 2006-year Altium BV                Serial# 00000000
```

Related information

-

Make utility option: -W

Command line syntax

`-W target`

Description

With this option the make utility considers the specified target file always as up to date and will not rebuild it.

Example

```
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

-

Make utility option: -w

Command line syntax

`-w`

Description

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

This option is only useful on UNIX systems.

Example

```
mkpcp -w
```

The make utility sends messages to standard out instead of standard error.

Related information

-

Make utility option: -x

Command line syntax

-x

Description

With this option the make utility shows extended error messages. Extended error messages give more detailed information about the exit status of the make utility after errors.

Example

```
mkpcp -x
```

If errors occur, the make utility gives extended information.

Related information

-

8.6. Parallel Make Utility Options

When you build a project in Eclipse, Eclipse generates a makefile and uses the make utility **amk** to build all your files. However, you can also use the make utility directly from the command line to build your project.

The invocation syntax is:

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

This section describes all options for the parallel make utility.

For detailed information about the parallel make utility and using makefiles see [Section 6.3, Make Utility amk](#).

Parallel make utility option: -?

Command line syntax

-?

Description

Displays an overview of all command line options.

Example

The following invocation displays a list of the available command line options:

```
amk -?
```

Related information

-

Parallel make utility option: -a

Command line syntax

-a

Description

Normally the make utility rebuilds only those files that are out of date. With this option you tell the make utility to rebuild *all* files, without checking whether they are out of date.

Example

```
amk -a
```

Rebuilds all your files, regardless of whether they are out of date or not.

Related information

-

Parallel make utility option: -f

Command line syntax

```
-f my_makefile
```

Description

By default the make utility uses the file `makefile` to build your files.

With this option you tell the make utility to use the specified file instead of the file `makefile`. Multiple `-f` options act as if all the makefiles were concatenated in a left-to-right order.

If you use `'-'` instead of a makefile name it means that the information is read from `stdin`.

Example

```
amk -f mymake
```

The make utility uses the file `mymake` to build your files.

Related information

-

Parallel make utility option: -G

Command line syntax

`-G path`

Description

Normally you must call the make utility from the directory where your makefile and other files are stored.

With the option **-G** you can call the make utility from within another directory. The *path* is the path to the directory where your makefile and other files are stored and can be absolute or relative to your current directory.

The macro `SUBDIR` is defined with the value of *path*.

Example

Suppose your makefile and other files are stored in the directory `..\myfiles`. You can call the make utility, for example, as follows:

```
amk -G ..\myfiles
```

Related information

-

Parallel make utility option: -j / -J

Menu

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

The Properties dialog appears.

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

In the right pane the C/C++ Build page appears.

3. On the Behaviour tab, select **Use parallel build**.

4. You can specify the number of parallel jobs, or you can use an optimal number of jobs. In the last case, **amk** will fork as many jobs in parallel as cores are available.

Command line syntax

`-j[number]`

`-J[number]`

Description

When these options you can limit the number of parallel jobs. The default is 1. Zero means no limit. When you omit the *number*, **amk** uses the number of cores detected.

Option **-J** is the same as **-j**, except that the number of parallel jobs is limited by the number of cores detected.

Example

```
amk -j3
```

Limit the number of parallel jobs to 3.

Related information

-

Parallel make utility option: -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

```
amk -k
```

If the make utility encounters an error, it stops building the current target but proceeds with the other targets that are defined in the makefile.

Related information

-

Parallel make utility option: -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

```
amk -n
```

The make utility does not perform any tasks but displays what it would do if called without the option `-n`.

Related information

[Parallel make utility option -s](#) (Do not print commands before execution)

Parallel make utility option: **-s**

Command line syntax

-s

Description

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

Example

```
amk -s
```

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

Related information

[Parallel make utility option **-n**](#) (Perform a dry run)

Parallel make utility option: -V

Command line syntax

`-V`

Description

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

Related information

-

8.7. 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.

For detailed information about the archiver, see [Section 6.4, Archiver](#).

Short and long option names

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

Overview of the options of the archiver utility

The following archiver options are available:

Description	Option	Sub-option
Main functions (key options)		
Replace or add an object module	-r	-a -b -c -u -v
Extract an object module from the library	-x	-o -v
Delete object module from library	-d	-v
Move object module to another position	-m	-a -b -v
Print a table of contents of the library	-t	-s0 -s1
Print object module to standard output	-p	
Sub-options		
Append or move new modules after existing module <i>name</i>	-a name	
Append or move new modules before existing module <i>name</i>	-b name	
Create library without notification if library does not exist	-c	
Preserve last-modified date from the library	-o	
Print symbols in library modules	-s{0 1}	
Replace only newer modules	-u	
Verbose	-v	
Miscellaneous		

Description	Option	Sub-option
Display options	-?	
Display version header	-V	
Read options from <i>file</i>	-f <i>file</i>	
Suppress warnings above level <i>n</i>	-wn	

Archiver option: **--delete (-d)**

Command line syntax

```
--delete [--verbose]
```

```
-d [-v]
```

Description

Delete the specified object modules from a library. With the suboption **--verbose (-v)** the archiver shows which files are removed.

--verbose	-v	Verbose: the archiver shows which files are removed.
------------------	-----------	--

Example

```
arpcp --delete mylib.a obj1.o obj2.o
```

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

```
arpcp -d -v mylib.a obj1.o obj2.o
```

The archiver deletes `obj1.o` and `obj2.o` from the library `mylib.a` and displays which files are removed.

Related information

-

Archiver option: `--dump (-p)`

Command line syntax

`--dump`

`-p`

Description

Print the specified object module(s) in the library to standard output.

This option is only useful when you redirect or pipe the output to other files or tools that serve your own purposes. Normally you do not need this option.

Example

```
arpcp --dump mylib.a obj1.o > file.o
```

The archiver prints the file `obj1.o` to standard output where it is redirected to the file `file.o`. The effect of this example is very similar to extracting a file from the library but in this case the 'extracted' file gets another name.

Related information

-

Archiver option: **--extract (-x)**

Command line syntax

```
--extract [--modtime] [--verbose]
```

```
-x [-o] [-v]
```

Description

Extract an existing module from the library.

--modtime	-o	Give the extracted object module the same date as the last-modified date that was recorded in the library. Without this suboption it receives the last-modified date of the moment it is extracted.
--verbose	-v	Verbose: the archiver shows which files are extracted.

Example

To extract the file `obj1.o` from the library `mylib.a`:

```
arpcp --extract mylib.a obj1.o
```

If you do not specify an object module, all object modules are extracted:

```
arpcp -x mylib.a
```

Related information

-

Archiver option: --help (-?)

Command line syntax

```
--help[=item]
```

```
-?
```

You can specify the following argument:

options	Show extended option descriptions
----------------	-----------------------------------

Description

Displays an overview of all command line options. When you specify the argument **options** you can list detailed option descriptions.

Example

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

```
arpcp -?  
arpcp --help  
arpcp
```

To see a detailed description of the available options, enter:

```
arpcp --help=options
```

Related information

-

Archiver option: --move (-m)

Command line syntax

```
--move [-a posname] [-b posname]
```

```
-m [-a posname] [-b posname]
```

Description

Move the specified object modules to another position in the library.

The ordering of members in a library can make a difference in how programs are linked if a symbol is defined in more than one member.

By default, the specified members are moved to the end of the archive. Use the suboptions **-a** or **-b** to move them to a specified place instead.

--after=<i>posname</i>	-a	Move the specified object module(s) after the existing module <i>posname</i> .
--before=<i>posname</i>	-b	Move the specified object module(s) before the existing module <i>posname</i> .

Example

Suppose the library `mylib.a` contains the following objects (see option **--print**):

```
obj1.o  
obj2.o  
obj3.o
```

To move `obj1.o` to the end of `mylib.a`:

```
arpcp --move mylib.a obj1.o
```

To move `obj3.o` just before `obj2.o`:

```
arpcp -m -b obj3.o mylib.a obj2.o
```

The library `mylib.a` after these two invocations now looks like:

```
obj3.o  
obj2.o  
obj1.o
```

Related information

Archiver option **--print (-t)** (Print library contents)

Archiver option: **--option-file (-f)**

Command line syntax

```
--option-file=file
```

```
-f file
```

Description

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

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

You can specify the option **--option-file (-f)** multiple times.

If you use '-' instead of a filename it means that the options are read from `stdin`.

Format of an option file

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

```
-x mylib.a obj1.o  
-w5
```

TASKING VX-toolset for PCP User Guide

Specify the option file to the archiver:

```
arpcp --option-file=myoptions
```

This is equivalent to the following command line:

```
arpcp -x mylib.a obj1.o -w5
```

Related information

-

Archiver option: --print (-t)

Command line syntax

```
--print [--symbols=0|1]
```

```
-t [-s0|-s1]
```

Description

Print a table of contents of the library to standard output. With the suboption **-s0** the archiver displays all symbols per object file.

--symbols=0	-s0	Displays per object the name of the object itself and all symbols in the object.
--symbols=1	-s1	Displays the symbols of all object files in the library in the form <i>library_name:object_name:symbol_name</i>

Example

```
arpcp --print mylib.a
```

The archiver prints a list of all object modules in the library `mylib.a`:

```
arpcp -t -s0 mylib.a
```

The archiver prints per object all symbols in the library. For example:

```
cstart.o
  symbols:
    _PCP__context_8
    _PCP__cstart
    _PCP_channel_2
printf.o
  symbols:
    _PCP__data__printf
    _PCP_printf
```

Related information

-

Archiver option: --replace (-r)

Command line syntax

```
--replace [--after=posname] [--before=posname][--create] [--newer-only] [--verbose]
-r [-a posname] [-b posname][-c] [-u] [-v]
```

Description

You can use the option **--replace (-r)** for several purposes:

- Adding new objects to the library
- Replacing objects in the library with the same object of a newer date
- Creating a new library

The option **--replace (-r)** normally *adds* a new module to the library. However, if the library already contains a module with the specified name, the existing module is *replaced*. If you specify a library that does not exist, the archiver creates a new library with the specified name.

If you add a module to the library without specifying the suboption **-a** or **-b**, the specified module is added at the end of the archive. Use the suboptions **-a** or **-b** to insert them after/before a specified place instead.

--after=posname	-a	Insert the specified object module(s) after the existing module <i>posname</i> .
--before=posname	-b	Insert the specified object module(s) before the existing module <i>posname</i> .
--create	-c	Create a new library without checking whether it already exists. If the library already exists, it is overwritten.
--newer-only	-u	Insert the specified object module only if it is newer than the module in the library.
--verbose	-v	Verbose: the archiver shows which files are replaced.

The suboptions **-a** or **-b** have no effect when an object is added to the library.

Example

Suppose the library `mylib.a` contains the following object (see option **--print**):

```
obj1.o
```

To add `obj2.o` to the end of `mylib.a`:

```
arpcp --replace mylib.a obj2.o
```

To insert `obj3.o` just before `obj2.o`:

```
arpcp -r -b obj2.o mylib.a obj3.o
```

The library `mylib.a` after these two invocations now looks like:

```
obj1.o  
obj3.o  
obj2.o
```

Creating a new library

To *create a new library file*, add an object file and specify a library that does not yet exist:

```
arpcp --replace obj1.o newlib.a
```

The archiver creates the library `newlib.a` and adds the object `obj1.o` to it.

To *create a new library file and overwrite an existing library*, add an object file and specify an existing library with the supoption `-c`:

```
arpcp -r -c obj1.o mylib.a
```

The archiver overwrites the library `mylib.a` and adds the object `obj1.o` to it. The new library `mylib.a` only contains `obj1.o`.

Related information

Archiver option `--print (-t)` (Print library contents)

Archiver option: **--version (-V)**

Command line syntax

`--version`

`-V`

Description

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

Example

```
arpcp -V
```

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

```
TASKING VX-toolset for PCP: ELF archiver    vx.yrz Build nnn  
Copyright 2006-year Altium BV              Serial# 00000000
```

Related information

-

Archiver option: **--warning (-w)**

Command line syntax

```
--warning=level
```

```
-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 suppress warnings above level 5:

```
arpcp --extract --warning=5 mylib.a obj1.o
```

Related information

-

Chapter 9. Libraries

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

[Section 9.1, *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.

[Section 9.2, *C Library Reentrancy*](#), gives an overview of which functions are reentrant and which are not.

The following libraries are included in the PCP toolset. Both Eclipse and the control program **ccpcp** automatically select the appropriate libraries depending on the specified options.

C library

Libraries	Description
libc[f].a	C libraries Optional letter: f = library compiled for <code>__far</code> memory
libfp[tf].a	Floating-point libraries Optional letter: t = trapping (control program option --fp-trap) f = library compiled for <code>__far</code> memory

9.1. 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 the debugger to simulate an I/O environment which enables you to debug your application.

9.1.1. `assert.h`

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

9.1.2. `complex.h`

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

The header file `complex.h` also defines the following macros for backward compatibility:

```
complex    _Complex    /* C99 keyword */
imaginary  _Imaginary  /* C99 keyword */
```

Parallel sets of functions are defined for double, float and long double. They are respectively named *function*, *functionf*, *functionl*. All long type functions, though declared in `complex.h`, are implemented as the `double` type variant which nearly always meets the requirement in embedded applications.

This implementation uses the obvious implementation for complex multiplication; and a more sophisticated implementation for division and absolute value calculations which handles underflow, overflow and infinities with more care. The ISO C99 `#pragma CX_LIMITED_RANGE` therefore has no effect.

Trigonometric functions

<code>csin</code>	<code>csinf</code>	<code>csinl</code>	Returns the complex sine of z .
<code>ccos</code>	<code>ccosf</code>	<code>ccosl</code>	Returns the complex cosine of z .
<code>ctan</code>	<code>ctanf</code>	<code>ctanl</code>	Returns the complex tangent of z .
<code>casin</code>	<code>casinf</code>	<code>casinl</code>	Returns the complex arc sine $\sin^{-1}(z)$.
<code>cacos</code>	<code>cacosf</code>	<code>cacosl</code>	Returns the complex arc cosine $\cos^{-1}(z)$.
<code>catan</code>	<code>catanf</code>	<code>catanl</code>	Returns the complex arc tangent $\tan^{-1}(z)$.
<code>csinh</code>	<code>csinhf</code>	<code>csinhl</code>	Returns the complex hyperbolic sine of z .
<code>ccosh</code>	<code>ccoshf</code>	<code>ccoshl</code>	Returns the complex hyperbolic cosine of z .
<code>ctanh</code>	<code>ctanhf</code>	<code>ctanhl</code>	Returns the complex hyperbolic tangent of z .
<code>casinh</code>	<code>casinhf</code>	<code>casinhl</code>	Returns the complex arc hyperbolic sinus of z .
<code>cacosh</code>	<code>cacoshf</code>	<code>cacoshl</code>	Returns the complex arc hyperbolic cosine of z .
<code>catanh</code>	<code>catanhf</code>	<code>catanhl</code>	Returns the complex arc hyperbolic tangent of z .

Exponential and logarithmic functions

<code>cexp</code>	<code>cexpf</code>	<code>cexpl</code>	Returns the result of the complex exponential function e^z .
<code>clog</code>	<code>clogf</code>	<code>clogl</code>	Returns the complex natural logarithm.

Power and absolute-value functions

<code>cabs</code>	<code>cabsf</code>	<code>cabsl</code>	Returns the complex absolute value of z (also known as <i>norm</i> , <i>modulus</i> or <i>magnitude</i>).
<code>cpow</code>	<code>cpowf</code>	<code>cpowl</code>	Returns the complex value of x raised to the power y (x^y) where both x and y are complex numbers.
<code>csqrt</code>	<code>csqrtf</code>	<code>csqrtl</code>	Returns the complex square root of z .

Manipulation functions

<code>carg</code>	<code>cargf</code>	<code>cargl</code>	Returns the argument of z (also known as <i>phase angle</i>).
<code>cimag</code>	<code>cimagf</code>	<code>cimagl</code>	Returns the imaginary part of z as a real (respectively as a double, float, long double)
<code>conj</code>	<code>conjf</code>	<code>conjl</code>	Returns the complex conjugate value (the sign of its imaginary part is reversed).

<code>cproj</code>	<code>cprojf</code>	<code>cprojl</code>	Returns the value of the projection of <code>z</code> onto the Riemann sphere.
<code>creal</code>	<code>crealf</code>	<code>creall</code>	Returns the real part of <code>z</code> as a real (respectively as a double, float, long double)

9.1.3. `cstart.h`

The header file `cstart.h` controls the system startup code's general settings and register initializations. It contains defines only, no functions.

9.1.4. `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.

<code>ctype.h</code>	<code>wctype.h</code>	Description
<code>isalnum</code>	<code>iswalnum</code>	Returns a non-zero value when <code>c</code> is an alphabetic character or a number (<code>[A-Z][a-z][0-9]</code>).
<code>isalpha</code>	<code>iswalpha</code>	Returns a non-zero value when <code>c</code> is an alphabetic character (<code>[A-Z][a-z]</code>).
<code>isblank</code>	<code>iswblank</code>	Returns a non-zero value when <code>c</code> is a blank character (tab, space...)
<code>iscntrl</code>	<code>iswcntrl</code>	Returns a non-zero value when <code>c</code> is a control character.
<code>isdigit</code>	<code>iswdigit</code>	Returns a non-zero value when <code>c</code> is a numeric character (<code>[0-9]</code>).
<code>isgraph</code>	<code>iswgraph</code>	Returns a non-zero value when <code>c</code> is printable, but not a space.
<code>islower</code>	<code>iswlower</code>	Returns a non-zero value when <code>c</code> is a lowercase character (<code>[a-z]</code>).
<code>isprint</code>	<code>iswprint</code>	Returns a non-zero value when <code>c</code> is printable, including spaces.
<code>ispunct</code>	<code>iswpunct</code>	Returns a non-zero value when <code>c</code> is a punctuation character (such as <code>','</code> , <code>','</code> , <code>!</code>).
<code>isspace</code>	<code>iswspace</code>	Returns a non-zero value when <code>c</code> is a space type character (space, tab, vertical tab, formfeed, linefeed, carriage return).
<code>isupper</code>	<code>iswupper</code>	Returns a non-zero value when <code>c</code> is an uppercase character (<code>[A-Z]</code>).
<code>isxdigit</code>	<code>iswxdigit</code>	Returns a non-zero value when <code>c</code> is a hexadecimal digit (<code>[0-9][A-F][a-f]</code>).
<code>tolower</code>	<code>towlower</code>	Returns <code>c</code> converted to a lowercase character if it is an uppercase character, otherwise <code>c</code> is returned.
<code>toupper</code>	<code>toupper</code>	Returns <code>c</code> converted to an uppercase character if it is a lowercase character, otherwise <code>c</code> is returned.
<code>_tolower</code>	-	Converts <code>c</code> to a lowercase character, does not check if <code>c</code> really is an uppercase character. Implemented as macro. This macro function is not defined in ISO C99.
<code>_toupper</code>	-	Converts <code>c</code> to an uppercase character, does not check if <code>c</code> really is a lowercase character. Implemented as macro. This macro function is not defined in ISO C99.

ctype.h	wctype.h	Description
<code>isascii</code>		Returns a non-zero value when <code>c</code> is in the range of 0 and 127. This function is not defined in ISO C99.
<code>toascii</code>		Converts <code>c</code> to an ASCII value (strip highest bit). This function is not defined in ISO C99.

9.1.5. `dbg.h`

The header file `dbg.h` contains the debugger call interface for file system simulation. It contains low level functions. This header file is not defined in ISO C99.

<code>_dbg_trap</code>	Low level function to trap debug events
<code>_argcv(const char *buf, size_t size)</code>	Low level function for command line argument passing

9.1.6. `errno.h`

`int errno` External variable that holds implementation defined error codes.

The following error codes are defined as macros in `errno.h`:

<code>EPERM</code>	1	Operation not permitted
<code>ENOENT</code>	2	No such file or directory
<code>EINTR</code>	3	Interrupted system call
<code>EIO</code>	4	I/O error
<code>EBADF</code>	5	Bad file number
<code>EAGAIN</code>	6	No more processes
<code>ENOMEM</code>	7	Not enough core
<code>EACCES</code>	8	Permission denied
<code>EFAULT</code>	9	Bad address
<code>EEXIST</code>	10	File exists
<code>ENOTDIR</code>	11	Not a directory
<code>EISDIR</code>	12	Is a directory
<code>EINVAL</code>	13	Invalid argument
<code>ENFILE</code>	14	File table overflow
<code>EMFILE</code>	15	Too many open files
<code>ETXTBSY</code>	16	Text file busy
<code>ENOSPC</code>	17	No space left on device
<code>ESPIPE</code>	18	Illegal seek
<code>EROFS</code>	19	Read-only file system
<code>EPIPE</code>	20	Broken pipe
<code>ELOOP</code>	21	Too many levels of symbolic links
<code>ENAMETOOLONG</code>	22	File name too long

Floating-point errors

<code>EDOM</code>	23	Argument too large
<code>ERANGE</code>	24	Result too large

Errors returned by printf/scanf

ERR_FORMAT	25	Illegal format string for printf/scanf
ERR_NOFLOAT	26	Floating-point not supported
ERR_NOLONG	27	Long not supported
ERR_NOPOINT	28	Pointers not supported

Encoding errors set by functions like fgetc, getc, mbtrowc, etc ...

EILSEQ	29	Invalid or incomplete multibyte or wide character
--------	----	---

Errors returned by RTOS

ECANCELED	30	Operation canceled
ENODEV	31	No such device

9.1.7. 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 C99.

`open` Opens a file a file for reading or writing. Calls `_open`.
(FSS implementation)

9.1.8. fenv.h

Contains mechanisms to control the floating-point environment. The functions in this header file are not implemented.

<code>fegetenv</code>	Stores the current floating-point environment. <i>(Not implemented)</i>
<code>feholdexcept</code>	Saves the current floating-point environment and installs an environment that ignores all floating-point exceptions. <i>(Not implemented)</i>
<code>fesetenv</code>	Restores a previously saved (<code>fegetenv</code> or <code>feholdexcept</code>) floating-point environment. <i>(Not implemented)</i>
<code>feupdateenv</code>	Saves the currently raised floating-point exceptions, restores a previously saved floating-point environment and finally raises the saved exceptions. <i>(Not implemented)</i>
<code>feclearexcept</code>	Clears the current exception status flags corresponding to the flags specified in the argument. <i>(Not implemented)</i>
<code>fegetexceptflag</code>	Stores the current setting of the floating-point status flags. <i>(Not implemented)</i>
<code>feraiseexcept</code>	Raises the exceptions represented in the argument. As a result, other exceptions may be raised as well. <i>(Not implemented)</i>
<code>fesetexceptflag</code>	Sets the current floating-point status flags. <i>(Not implemented)</i>

TASKING VX-toolset for PCP User Guide

`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.

9.1.9. float.h

The header file `float.h` defines the characteristics of the real floating-point types `float`, `double` and `long double`.

`float.h` used to contain prototypes for the functions `copysign(f)`, `isinf(f)`, `isfinite(f)`, `isnan(f)` and `scalb(f)`. These functions have accordingly to the ISO C99 standard been moved to the header file `math.h`. See also [Section 9.1.16, `math.h` and `tgmath.h`](#).

The following functions are only available for ISO C90:

```
copysignf(float f, float s) Copies the sign of the second argument s to the value of the first
                           argument f and returns the result.
copysign(double d, double s) Copies the sign of the second argument s to the value of the first
                              argument d and returns the result.
isinff(float f)             Test the variable f on being an infinite (IEEE-754) value.
isinf(double d);           Test the variable d on being an infinite (IEEE-754) value.
isfinitef(float f)         Test the variable f on being a finite (IEEE-754) value.
isfinite(double d)         Test the variable d on being a finite (IEEE-754) value.
isnanf(float f)            Test the variable f on being NaN (Not a Number, IEEE-754) .
isnan(double d)            Test the variable d on being NaN (Not a Number, IEEE-754) .
scalbf(float f, int p)     Returns f * 2^p for integral values without computing 2^N.
scalb(double d, int p)     Returns d * 2^p for integral values without computing 2^N. (See
                           also scalbn in Section 9.1.16, math.h and tgmath.h)
```

9.1.10. 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 C99 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.

<code>imaxabs(intmax_t j)</code>	Returns the absolute value of <code>j</code>
<code>imaxdiv(intmax_t numer, intmax_t denom)</code>	Computes <code>numer/denom</code> and <code>numer % denom</code> . The result is stored in the <code>quot</code> and <code>rem</code> components of the <code>imaxdiv_t</code> structure type.
<code>strtoimax(const char * restrict nptr, char ** restrict endptr, int base)</code>	Convert string to maximum sized integer. (Compare <code>strtoll</code>)
<code>strtoumax(const char * restrict nptr, char ** restrict endptr, int base)</code>	Convert string to maximum sized unsigned integer. (Compare <code>strtoull</code>)
<code>wcstoimax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base)</code>	Convert wide string to maximum sized integer. (Compare <code>wcstoll</code>)
<code>wcstoumax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base)</code>	Convert wide string to maximum sized unsigned integer. (Compare <code>wcstoull</code>)

9.1.11. io.h

The header file `io.h` contains prototypes for low level I/O functions. This header file is not defined in ISO C99.

<code>_close(fd)</code>	Used by the functions <code>close</code> and <code>fclose</code> . (<i>FSS implementation</i>)
<code>_lseek(fd, offset, whence)</code>	Used by all file positioning functions: <code>fgetpos</code> , <code>fseek</code> , <code>fsetpos</code> , <code>ftell</code> , <code>rewind</code> . (<i>FSS implementation</i>)
<code>_open(fd, flags)</code>	Used by the functions <code>fopen</code> and <code>freopen</code> . (<i>FSS implementation</i>)
<code>_read(fd, *buff, cnt)</code>	Reads a sequence of characters from a file. (<i>FSS implementation</i>)
<code>_unlink(*name)</code>	Used by the function <code>remove</code> . (<i>FSS implementation</i>)
<code>_write(fd, *buffer, cnt)</code>	Writes a sequence of characters to a file. (<i>FSS implementation</i>)

9.1.12. 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  ^=
```

9.1.13. limits.h

Contains the sizes of integral types, defined as macros.

9.1.14. locale.h

To keep C code reasonable portable across different languages and cultures, a number of facilities are provided in the header file `local.h`.

```
char *setlocale( int category, const char *locale )
```

The function above changes locale-specific features of the run-time library as specified by the category to change and the name of the locale.

The following categories are defined and can be used as input for this function:

LC_ALL	0	LC_NUMERIC	3
LC_COLLATE	1	LC_TIME	4
LC_CTYPE	2	LC_MONETARY	5

```
struct lconv *localeconv( void )
```

Returns a pointer to type `struct lconv` with values appropriate for the formatting of numeric quantities according to the rules of the current locale. The `struct lconv` in this header file is conforming the ISO standard.

9.1.15. malloc.h

The header file `malloc.h` contains prototypes for memory allocation functions. This include file is not defined in ISO C99, it is included for backwards compatibility with ISO C90. For ISO C99, the memory allocation functions are part of `stdlib.h`. See [Section 9.1.24, `stdlib.h` and `wchar.h`](#).

<code>malloc(size)</code>	Allocates space for an object with size <code>size</code> . The allocated space is not initialized. Returns a pointer to the allocated space.
<code>calloc(nobj, size)</code>	Allocates space for <code>n</code> objects with size <code>size</code> . The allocated space is initialized with zeros. Returns a pointer to the allocated space.
<code>free(*ptr)</code>	Deallocates the memory space pointed to by <code>ptr</code> which should be a pointer earlier returned by the <code>malloc</code> or <code>calloc</code> function.

`realloc(*ptr, size)`

Deallocates the old object pointed to by *ptr* and returns a pointer to a new object with size *size*, while preserving its contents. If the new size is smaller than the old size, some contents at the end of the old region will be discarded. If the new size is larger than the old size, all of the old contents are preserved and any bytes in the new object beyond the size of the old object will have indeterminate values.

9.1.16. `math.h` and `tgmath.h`

The header file `math.h` contains the prototypes for many mathematical functions. Before ISO C99, all functions were computed using the double type (the float was automatically converted to double, prior to calculation). In this ISO 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 and hyperbolic functions

math.h		tgmath.h		Description
<code>sin</code>	<code>sinf</code>	<code>sinl</code>	<code>sin</code>	Returns the sine of <i>x</i> .
<code>cos</code>	<code>cosf</code>	<code>cosl</code>	<code>cos</code>	Returns the cosine of <i>x</i> .
<code>tan</code>	<code>tanf</code>	<code>tanl</code>	<code>tan</code>	Returns the tangent of <i>x</i> .
<code>asin</code>	<code>asinf</code>	<code>asinl</code>	<code>asin</code>	Returns the arc sine $\sin^{-1}(x)$ of <i>x</i> .
<code>acos</code>	<code>acosf</code>	<code>acosl</code>	<code>acos</code>	Returns the arc cosine $\cos^{-1}(x)$ of <i>x</i> .
<code>atan</code>	<code>atanf</code>	<code>atanl</code>	<code>atan</code>	Returns the arc tangent $\tan^{-1}(x)$ of <i>x</i> .
<code>atan2</code>	<code>atan2f</code>	<code>atan2l</code>	<code>atan2</code>	Returns the result of: $\tan^{-1}(y/x)$.
<code>sinh</code>	<code>sinhf</code>	<code>sinhl</code>	<code>sinh</code>	Returns the hyperbolic sine of <i>x</i> .
<code>cosh</code>	<code>coshf</code>	<code>coshl</code>	<code>cosh</code>	Returns the hyperbolic cosine of <i>x</i> .
<code>tanh</code>	<code>tanhf</code>	<code>tanh1</code>	<code>tanh</code>	Returns the hyperbolic tangent of <i>x</i> .
<code>asinh</code>	<code>asinhf</code>	<code>asinh1</code>	<code>asinh</code>	Returns the arc hyperbolic sine of <i>x</i> .
<code>acosh</code>	<code>acoshf</code>	<code>acosh1</code>	<code>acosh</code>	Returns the non-negative arc hyperbolic cosine of <i>x</i> .
<code>atanh</code>	<code>atanhf</code>	<code>atanh1</code>	<code>atanh</code>	Returns the arc hyperbolic tangent of <i>x</i> .

Exponential and logarithmic functions

All of these functions are new in ISO C99, except for `exp`, `log` and `log10`.

math.h		tgmath.h		Description
<code>exp</code>	<code>expf</code>	<code>expl</code>	<code>exp</code>	Returns the result of the exponential function e^x .

math.h		tgmath.h		Description
exp2	exp2f	exp2l	exp2	Returns the result of the exponential function 2^x . (Not implemented)
expm1	expm1f	expm1l	expm1	Returns the result of the exponential function $e^x - 1$. (Not implemented)
log	logf	logl	log	Returns the natural logarithm $\ln(x)$, $x > 0$.
log10	log10f	log10l	log10	Returns the base-10 logarithm of x , $x > 0$.
log1p	log1pf	log1pl	log1p	Returns the base-e logarithm of $(1+x)$. $x < > -1$. (Not implemented)
log2	log2f	log2l	log2	Returns the base-2 logarithm of x . $x > 0$. (Not implemented)
ilogb	ilogbf	ilogbl	ilogb	Returns the signed exponent of x as an integer. $x > 0$. (Not implemented)
logb	logbf	logbl	logb	Returns the exponent of x as a signed integer in value in floating-point notation. $x > 0$. (Not implemented)

frexp, ldexp, modf, scalbn, scalbln

math.h		tgmath.h		Description
frexp	frexpf	frexpl	frexp	Splits a float x into fraction f and exponent n , so that: $f = 0.0$ or $0.5 \leq f \leq 1.0$ and $f \cdot 2^n = x$. Returns f , stores n .
ldexp	ldexpf	ldexpl	ldexp	Inverse of <code>frexp</code> . Returns the result of $x \cdot 2^n$. (x and n are both arguments).
modf	modff	modfl	-	Splits a float x into fraction f and integer n , so that: $ f < 1.0$ and $f + n = x$. Returns f , stores n .
scalbn	scalbnf	scalbnl	scalbn	Computes the result of $x \cdot \text{FLT_RADIX}^n$. efficiently, not normally by computing FLT_RADIX^n explicitly.
scalbln	scalblnf	scalblnl	scalbln	Same as <code>scalbn</code> but with argument n as long int.

Rounding functions

math.h		tgmath.h		Description
ceil	ceilf	ceill	ceil	Returns the smallest integer not less than x , as a double.
floor	floorf	floorl	floor	Returns the largest integer not greater than x , as a double.
rint	rintf	rintl	rint	Returns the rounded integer value as an int according to the current rounding direction. See <code>fenv.h</code> . (Not implemented)
lrint	lrintf	lrintl	lrint	Returns the rounded integer value as a long int according to the current rounding direction. See <code>fenv.h</code> . (Not implemented)
llrint	llrintf	llrintl	llrint	Returns the rounded integer value as a long long int according to the current rounding direction. See <code>fenv.h</code> . (Not implemented)

math.h				tgmath.h	Description
nearbyint	nearbyintf	nearbyintl	nearbyint		Returns the rounded integer value as a floating-point according to the current rounding direction. See <code>fenv.h</code> . (<i>Not implemented</i>)
round	roundf	roundl	round		Returns the nearest integer value of <code>x</code> as int. (<i>Not implemented</i>)
lround	lroundf	lroundl	lround		Returns the nearest integer value of <code>x</code> as long int. (<i>Not implemented</i>)
llround	llroundf	llroundl	llround		Returns the nearest integer value of <code>x</code> as long long int. (<i>Not implemented</i>)
trunc	truncf	truncl	trunc		Returns the truncated integer value <code>x</code> . (<i>Not implemented</i>)

Remainder after division

math.h				tgmath.h	Description
fmod	fmodf	fmodl	fmod		Returns the remainder <code>r</code> of <code>x-ny</code> . <code>n</code> is chosen as <code>trunc(x/y)</code> . <code>r</code> has the same sign as <code>x</code> .
remainder	remainderf	remainderl	remainder		Returns the remainder <code>r</code> of <code>x-ny</code> . <code>n</code> is chosen as <code>trunc(x/y)</code> . <code>r</code> may not have the same sign as <code>x</code> . (<i>Not implemented</i>)
remquo	remquof	remquol	remquo		Same as <code>remainder</code> . In addition, the argument <code>*quo</code> is given a specific value (see ISO). (<i>Not implemented</i>)

Power and absolute-value functions

math.h				tgmath.h	Description
cbrt	cbrtf	cbrtl	cbrt		Returns the real cube root of <code>x</code> ($=x^{1/3}$). (<i>Not implemented</i>)
fabs	fabsf	fabsl	fabs		Returns the absolute value of <code>x</code> ($ x $). (<code>abs</code> , <code>labs</code> , <code>llabs</code> , <code>div</code> , <code>ldiv</code> , <code>lldiv</code> are defined in <code>stdlib.h</code>)
fma	fmaf	fmal	fma		Floating-point multiply add. Returns <code>x*y+z</code> . (<i>Not implemented</i>)
hypot	hypotf	hypotl	hypot		Returns the square root of x^2+y^2 .
pow	powf	powl	power		Returns <code>x</code> raised to the power <code>y</code> (x^y).
sqrt	sqrtf	sqrtl	sqrt		Returns the non-negative square root of <code>x</code> . <code>x >= 0</code> .

Manipulation functions: copysign, nan, nextafter, nexttoward

math.h				tgmath.h	Description
copysign	copysignf	copysignll	copysign		Returns the value of <code>x</code> with the sign of <code>y</code> .
nan	nanf	nanl	-		Returns a quiet NaN, if available, with content indicated through <code>tagp</code> . (<i>Not implemented</i>)

math.h	tgmath.h	Description
nextafter nextafterf nextafterl nextafter		Returns the next representable value in the specified format after x in the direction of y . Returns y if $x=y$. (Not implemented)
nexttoward nexttowardf nexttowardl nexttoward		Same as <code>nextafter</code> , except that the second argument in all three variants is of type long double. Returns y if $x=y$. (Not implemented)

Positive difference, maximum, minimum

math.h	tgmath.h	Description
fdim fdimf fdiml fdim		Returns the positive difference between: $ x-y $. (Not implemented)
fmax fmaxf fmaxl fmax		Returns the maximum value of their arguments. (Not implemented)
fmin fminf fminl fmin		Returns the minimum value of their arguments. (Not implemented)

Error and gamma (Not implemented)

math.h	tgmath.h	Description
erf erff erfl erf		Computes the error function of x . (Not implemented)
erfc erfcf erfcl erc		Computes the complementary error function of x . (Not implemented)
lgamma lgammaf lgammal lgamma		Computes the $\ast \log_e \Gamma(x) $ (Not implemented)
tgamma tgammaf tgammaal tgamma		Computes $\Gamma(x)$ (Not implemented)

Comparison macros

The next are implemented as macros. For any ordered pair of numeric values exactly one of the relationships - *less*, *greater*, and *equal* - is true. These macros are type generic and therefore do not have a parallel function in `tgmath.h`. All arguments must be expressions of real-floating type.

math.h	tgmath.h	Description
isgreater	-	Returns the value of $(x) > (y)$
isgreaterequal	-	Returns the value of $(x) >= (y)$
isless	-	Returns the value of $(x) < (y)$
islessequal	-	Returns the value of $(x) <= (y)$
islessgreater	-	Returns the value of $(x) < (y) \ \ (x) > (y)$

math.h	tgmath.h	Description
<code>isunordered</code>	-	Returns 1 if its arguments are unordered, 0 otherwise.

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.

math.h	tgmath.h	Description
<code>fpclassify</code>	-	Returns the class of its argument: FP_INFINITE, FP_NAN, FP_NORMAL, FP_SUBNORMAL or FP_ZERO
<code>isfinite</code>	-	Returns a nonzero value if and only if its argument has a finite value
<code>isinf</code>	-	Returns a nonzero value if and only if its argument has an infinite value
<code>isnan</code>	-	Returns a nonzero value if and only if its argument has NaN value.
<code>isnormal</code>	-	Returns a nonzero value if and only if its argument has a normal value.
<code>signbit</code>	-	Returns a nonzero value if and only if its argument value is negative.

9.1.17. setjmp.h

The `setjmp` and `longjmp` in this header file implement a primitive form of non-local 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().
```

9.1.18. 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
```

TASKING VX-toolset for PCP User Guide

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.

9.1.19. 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`. `va_copy` is new in ISO C99. This header file contains the following macros:

<code>va_arg(va_list ap, type)</code>	Returns the value of the next argument in the variable argument list. It's return type has the type of the given argument <code>type</code> . A next call to this macro will return the value of the next argument.
<code>va_copy(va_list dest, va_list src)</code>	This macro duplicates the current state of <code>src</code> in <code>dest</code> , creating a second pointer into the argument list. After this call, <code>va_arg()</code> may be used on <code>src</code> and <code>dest</code> independently.
<code>va_end(va_list ap)</code>	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.
<code>va_start(va_list ap, lastarg)</code>	This macro initializes <code>ap</code> . After this call, each call to <code>va_arg()</code> will return the value of the next argument. In our implementation, <code>va_list</code> cannot contain any bit type variables. Also the given argument <code>lastarg</code> must be the last non-bit type argument in the list.

9.1.20. stdbool.h

This header file contains the following macro definitions. These names for boolean type and values are consistent with C++. You are allowed to `#undefine` or `redefine` the macros below.

```
#define bool                _Bool
#define true                1
#define false               0
#define __bool_true_false_are_defined 1
```


9.1.21. `stddef.h`

This header file defines the types for common use:

`ptrdiff_t` Signed integer type of the result of subtracting two pointers.
`size_t` Unsigned integral type of the result of the `sizeof` operator.
`wchar_t` Integer type to represent character codes in large character sets.

Besides these types, the following macros are defined:

`NULL` Expands to 0 (zero).
`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`.

9.1.22. `stdint.h`

See [Section 9.1.10](#), [inttypes.h](#) and [stdint.h](#)

9.1.23. `stdio.h` and `wchar.h`

Types

The header file `stdio.h` contains functions for performing input and output. A number of functions also have a parallel wide character function or macro, defined in `wchar.h`. The header file `wchar.h` also includes `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. A `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

<code>stdio.h</code>	Description
<code>NULL</code>	Expands to 0 (zero).
<code>BUFSIZ</code>	Size of the buffer used by the <code>setbuf/setvbuf</code> function: 512
<code>EOF</code>	End of file indicator. Expands to -1.

stdio.h	Description
WEOF	End of file indicator. Expands to U_INT_MAX (defined in <code>limits.h</code>) 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 <code>wchar.h</code>).
FOPEN_MAX	Number of files that can be opened simultaneously: 10
FILENAME_MAX	Maximum length of a filename: 100
_IOFBF	Expand to an integer expression, suitable for use as argument to the <code>setvbuf</code> function.
_IOLBF	
_IONBF	
L_tmpnam	Size of the string used to hold temporary file names: 8 (<code>tmpxxxxx</code>)
TMP_MAX	Maximum number of unique temporary filenames that can be generated: 0x8000
SEEK_CUR	Expand to an integer expression, suitable for use as the third argument to the <code>fseek</code> function.
SEEK_END	
SEEK_SET	
stderr	Expressions of type "pointer to FILE" that point to the FILE objects associated with standard error, input and output streams.
stdin	
stdout	

File access

stdio.h	Description
<code>fopen(name, mode)</code>	Opens a file for a given mode. Available modes are: "r" read; open text file for reading "w" write; create text file for writing; if the file already exists, its contents is discarded "a" append; open existing text file or create new text file for writing at end of file "r+" open text file for update; reading and writing "w+" create text file for update; previous contents if any is discarded "a+" append; open or create text file for update, writes at end of file <i>(FSS implementation)</i>
<code>fclose(name)</code>	Flushes the data stream and closes the specified file that was previously opened with <code>fopen</code> . <i>(FSS implementation)</i>
<code>fflush(name)</code>	If stream is an output stream, any buffered but unwritten data is written. Else, the effect is undefined. <i>(FSS implementation)</i>
<code>freopen(name, mode, stream)</code>	Similar to <code>fopen</code> , but rather than generating a new value of type FILE *, the existing value is associated with a new stream. <i>(FSS implementation)</i>
<code>setbuf(stream, buffer)</code>	If buffer is NULL, buffering is turned off for the stream. Otherwise, <code>setbuf</code> is equivalent to: <code>(void) setvbuf(stream, buffer, _IOFBF, BUFSIZ)</code> .

stdio.h	Description
<code>setvbuf(<i>stream</i>,<i>buffer</i>,<i>mode</i>,<i>size</i>)</code>	Controls buffering for the <i>stream</i> ; this function must be called before reading or writing. <i>Mode</i> can have the following values: _IOFBF causes full buffering _IOLBF causes line buffering of text files _IONBF causes no buffering. If <i>buffer</i> is not NULL, it will be used as a buffer; otherwise a buffer will be allocated. <i>size</i> determines the buffer size.

Formatted 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', 'hh', 'l', 'll', 'L', 'j', 'z' or 't'. 'h' indicates that the argument is to be treated as a `short` or `unsigned short`. 'hh' indicates that the argument is to be treated as a `char` or `unsigned char`. 'l' should be used if the argument is a `long integer`, 'll' for a `long long`. 'L' indicates that the argument is a `long double`. 'j' indicates a pointer to `intmax_t` or `uintmax_t`, 'z' indicates a pointer to `size_t` and 't' indicates a pointer to `ptrdiff_t`.

Flags, length specifier, period, precision and length modifier are optional, the conversion character is not. The conversion character must be one of the following, if a character following '%' is not in the list, the behavior is undefined:

Character Printed as	
d, i	int, signed decimal
o	int, unsigned octal
x, X	int, unsigned hexadecimal in lowercase or uppercase respectively
u	int, unsigned decimal
c	int, single character (converted to unsigned char)
s	char *, the characters from the string are printed until a NULL character is found. When the given precision is met before, printing will also stop
f	double
e, E	double
g, G	double
a, A	double
n	int *, the number of characters written so far is written into the argument. This should be a pointer to an integer in default memory. No value is printed.
p	pointer
%	No argument is converted, a '%' is printed.

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 preceded by `'h'` if the argument is a pointer to `short` rather than `int`, or by `'hh'` if the argument is a pointer to `char`, or by `'l'` (letter ell) if the argument is a pointer to `long` or by `'ll'` for a pointer to `long long`, `'j'` for a pointer to `intmax_t` or `uintmax_t`, `'z'` for a pointer to `size_t` or `'t'` for a pointer to `ptrdiff_t`. The conversion characters `e`, `f`, and `g` may be preceded by `'l'` if the argument is a pointer to `double` rather than `float`, and by `'L'` for a pointer to a `long double`.

- A conversion specifier. '*', maximum field width and length modifier are optional, the conversion character is not. The conversion character must be one of the following, if a character following '%' is not in the list, the behavior is undefined.

Length specifier and length modifier are optional, the conversion character is not. The conversion character must be one of the following, if a character following '%' is not in the list, the behavior is undefined.

Character Scanned as	
d	int, signed decimal.
i	int, the integer may be given octal (i.e. a leading 0 is entered) or hexadecimal (leading "0x" or "0X"), or just decimal.
o	int, unsigned octal.
u	int, unsigned decimal.
x	int, unsigned hexadecimal in lowercase or uppercase.
c	single character (converted to unsigned char).
s	char *, a string of non white space characters. The argument should point to an array of characters, large enough to hold the string and a terminating NULL character.
f, F	float
e, E	float
g, G	float
a, A	float
n	int *, the number of characters written so far is written into the argument. No scanning is done.
p	pointer; hexadecimal value which must be entered without 0x- prefix.
[...]	Matches a string of input characters from the set between the brackets. A NULL character is added to terminate the string. Specifying []... includes the ']' character in the set of scanning characters.
[^...]	Matches a string of input characters not in the set between the brackets. A NULL character is added to terminate the string. Specifying [^]... includes the ']' character in the set.
%	Literal '%', no assignment is done.

scanf conversion characters

stdio.h	wchar.h	Description
<code>fscanf(stream, format, ...)</code>	<code>fwscanf(stream, format, ...)</code>	Performs a formatted read from the given <i>stream</i> . Returns the number of items converted successfully. (FSS implementation)
<code>scanf(format, ...)</code>	<code>wscanf(format, ...)</code>	Performs a formatted read from <code>stdin</code> . Returns the number of items converted successfully. (FSS implementation)
<code>sscanf(*s, format, ...)</code>	<code>swscanf(*s, format, ...)</code>	Performs a formatted read from the string <i>s</i> . Returns the number of items converted successfully.

stdio.h	wchar.h	Description
<code>vfscanf(stream, format, arg)</code>	<code>vwscanf(stream, format, arg)</code>	Same as <code>fscanf/fwscanf</code> , but extra arguments are given as variable argument list <code>arg</code> . (See Section 9.1.19, <code>stdarg.h</code>)
<code>vscanf(format, arg)</code>	<code>vwscanf(format, arg)</code>	Same as <code>sscanf/swscanf</code> , but extra arguments are given as variable argument list <code>arg</code> . (See Section 9.1.19, <code>stdarg.h</code>)
<code>vsscanf(*s, format, arg)</code>	<code>vswscanf(*s, format, arg)</code>	Same as <code>scanf/wscanf</code> , but extra arguments are given as variable argument list <code>arg</code> . (See Section 9.1.19, <code>stdarg.h</code>)
<code>fprintf(stream, format, ...)</code>	<code>fwprintf(stream, format, ...)</code>	Performs a formatted write to the given <code>stream</code> . Returns EOF/WEOF on error. (FSS implementation)
<code>printf(format, ...)</code>	<code>wprintf(format, ...)</code>	Performs a formatted write to the stream <code>stdout</code> . Returns EOF/WEOF on error. (FSS implementation)
<code>sprintf(*s, format, ...)</code>		Performs a formatted write to string <code>s</code> . Returns EOF/WEOF on error.
<code>snprintf(*s, n, format, ...)</code>	<code>swprintf(*s, n, format, ...)</code>	Same as <code>sprintf</code> , but <code>n</code> specifies the maximum number of characters (including the terminating null character) to be written.
<code>vfprintf(stream, format, arg)</code>	<code>vwprintf(stream, format, arg)</code>	Same as <code>fprintf/fwprintf</code> , but extra arguments are given as variable argument list <code>arg</code> . (See Section 9.1.19, <code>stdarg.h</code>) (FSS implementation)
<code>vprintf(format, arg)</code>	<code>wprintf(format, arg)</code>	Same as <code>printf/wprintf</code> , but extra arguments are given as variable argument list <code>arg</code> . (See Section 9.1.19, <code>stdarg.h</code>) (FSS implementation)
<code>vsprintf(*s, format, arg)</code>	<code>vswprintf(*s, format, arg)</code>	Same as <code>sprintf/swprintf</code> , but extra arguments are given as variable argument list <code>arg</code> . (See Section 9.1.19, <code>stdarg.h</code>)

Character input/output

stdio.h	wchar.h	Description
<code>fgetc(stream)</code>	<code>fgetwc(stream)</code>	Reads one character from <code>stream</code> . Returns the read character, or EOF/WEOF on error. (FSS implementation)
<code>getc(stream)</code>	<code>getwc(stream)</code>	Same as <code>fgetc/fgetwc</code> except that is implemented as a macro. (FSS implementation) NOTE: Currently #defined as <code>getchar()/getwchar()</code> because FILE I/O is not supported. Returns the read character, or EOF/WEOF on error.

stdio.h	wchar.h	Description
<code>getchar(stdin)</code>	<code>getwchar(stdin)</code>	Reads one character from the <code>stdin</code> stream. Returns the character read or EOF/WEOF on error. Implemented as macro. (FSS implementation)
<code>fgets(*s, n, stream)</code>	<code>fgetws(*s, n, stream)</code>	Reads at most the next $n-1$ 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. (FSS implementation)
<code>gets(*s, n, stdin)</code>	-	Reads at most the next $n-1$ 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. (FSS implementation)
<code>ungetc(c, stream)</code>	<code>ungetwc(c, stream)</code>	Pushes character <code>c</code> back onto the input <code>stream</code> . Returns EOF/WEOF on error.
<code>fputc(c, stream)</code>	<code>fputwc(c, stream)</code>	Put character <code>c</code> onto the given <code>stream</code> . Returns EOF/WEOF on error. (FSS implementation)
<code>putc(c, stream)</code>	<code>putwc(c, stream)</code>	Same as <code>fputc/fputwc</code> except that is implemented as a macro. (FSS implementation)
<code>putchar(c, stdout)</code>	<code>putwchar(c, stdout)</code>	Put character <code>c</code> onto the <code>stdout</code> stream. Returns EOF/WEOF on error. Implemented as macro. (FSS implementation)
<code>fputs(*s, stream)</code>	<code>fputws(*s, stream)</code>	Writes string <code>s</code> to the given <code>stream</code> . Returns EOF/WEOF on error. (FSS implementation)
<code>puts(*s)</code>	-	Writes string <code>s</code> to the <code>stdout</code> stream. Returns EOF/WEOF on error. (FSS implementation)

Direct input/output

stdio.h	Description
<code>fread(ptr, size, nobj, stream)</code>	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. (FSS implementation)
<code>fwrite(ptr, size, nobj, stream)</code>	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. (FSS implementation)

Random access

stdio.h	Description
<code>fseek(stream, offset, origin)</code>	Sets the position indicator for <code>stream</code> . (FSS implementation)

When repositioning a binary file, the new position `origin` is given by the following macros:

TASKING VX-toolset for PCP User Guide

SEEK_SET 0 *offset* characters from the beginning of the file
SEEK_CUR 1 *offset* characters from the current position in the file
SEEK_END 2 *offset* characters from the end of the file

`ftell(stream)` Returns the current file position for *stream*, or -1L on error. (FSS implementation)

`rewind(stream)` Sets the file position indicator for the *stream* to the beginning of the file. This function is equivalent to:
(void) `fseek(stream, 0L, SEEK_SET);`
`clearerr(stream);`
(FSS implementation)

`fgetpos(stream, pos)` Stores the current value of the file position indicator for *stream* in the object pointed to by *pos*. (FSS implementation)

`fsetpos(stream, pos)` Positions *stream* at the position recorded by `fgetpos` in **pos*. (FSS implementation)

Operations on files

stdio.h	Description
<code>remove(<i>file</i>)</code>	Removes the named file, so that a subsequent attempt to open it fails. Returns a non-zero value if not successful.
<code>rename(<i>old</i>, <i>new</i>)</code>	Changes the name of the file from old name to new name. Returns a non-zero value if not successful.
<code>tmpfile()</code>	Creates a temporary file of the mode "wb+" that will be automatically removed when closed or when the program terminates normally. Returns a <code>file</code> pointer.
<code>tmpnam(<i>buffer</i>)</code>	Creates new file names that do not conflict with other file names currently in use. The new file name is stored in a <i>buffer</i> which must have room for <code>L_tmpnam</code> characters. Returns a pointer to the temporary name. The file names are created in the current directory and all start with "tmp". At most <code>TMP_MAX</code> unique file names can be generated.

Error handling

stdio.h	Description
<code>clearerr(<i>stream</i>)</code>	Clears the end of file and error indicators for <i>stream</i> .
<code>ferror(<i>stream</i>)</code>	Returns a non-zero value if the error indicator for <i>stream</i> is set.
<code>feof(<i>stream</i>)</code>	Returns a non-zero value if the end of file indicator for <i>stream</i> is set.
<code>perror(<i>*s</i>)</code>	Prints <i>s</i> and the error message belonging to the integer <code>errno</code> . (See Section 9.1.6, <code>errno.h</code>)

9.1.24. `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

<code>EXIT_SUCCESS</code>	Predefined exit codes that can be used in the <code>exit</code> function.
<code>0</code>	
<code>EXIT_FAILURE</code>	
<code>1</code>	
<code>RAND_MAX</code>	Highest number that can be returned by the <code>rand/srand</code> function.
<code>32767</code>	
<code>MB_CUR_MAX</code>	1 Maximum number of bytes in a multibyte character for the extended character set specified by the current locale (category <code>LC_CTYPE</code> , see Section 9.1.14, <i>locale.h</i>).

Numeric conversions

The following functions convert the initial portion of a string `*s` to a double, int, long int and long long int value respectively.

```
double    atof(*s)
int       atoi(*s)
long     atol(*s)
long long atoll(*s)
```

The following functions convert the initial portion of the string `*s` to a float, double and long double value respectively. `*endp` will point to the first character not used by the conversion.

stdlib.h		wchar.h	
float	<code>strtof(*s,**endp)</code>	float	<code>wcstof(*s,**endp)</code>
double	<code>strtod(*s,**endp)</code>	double	<code>wctod(*s,**endp)</code>
long double	<code>strtold(*s,**endp)</code>	long double	<code>wctold(*s,**endp)</code>

The following 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	wchar.h
long strtol (*s,**endp,base)	long wcstol (*s,**endp,base)
long long strtoll (*s,**endp,base)	long long wcstoll (*s,**endp,base)
unsigned long strtoul (*s,**endp,base)	unsigned long wcstoul (*s,**endp,base)
unsigned long long strtoull (*s,**endp,base)	unsigned long long wcstoull (*s,**endp,base)

Random number generation

- `rand` Returns a pseudo random integer in the range 0 to `RAND_MAX`.
- `srand(seed)` Same as `rand` but uses *seed* for a new sequence of pseudo random numbers.

Memory management

- `malloc(size)` Allocates space for an object with size *size*. The allocated space is not initialized. Returns a pointer to the allocated space.
- `calloc(nobj,size)` Allocates space for *n* objects with size *size*. The allocated space is initialized with zeros. Returns a pointer to the allocated space.
- `free(*ptr)` Deallocates the memory space pointed to by *ptr* which should be a pointer earlier returned by the `malloc` or `calloc` function.
- `realloc(*ptr,size)` Deallocates the old object pointed to by *ptr* and returns a pointer to a new object with size *size*, while preserving its contents. If the new size is smaller than the old size, some contents at the end of the old region will be discarded. If the new size is larger than the old size, all of the old contents are preserved and any bytes in the new object beyond the size of the old object will have indeterminate values.

Environment communication

- `abort()` Causes abnormal program termination. If the signal `SIGABRT` is caught, the signal handler may take over control. (See [Section 9.1.18, *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 int`, and `long long int` `j` respectively.
`long labs(j)`
`long long llabs(j)`

`div_t div(x,y)` Compute x/y and $x\%y$ in a single operation. `X` and `y` have respectively type `int`, `long int` and `long long int`. The result is stored in the members `ldiv_t ldiv(x,y)` `quot` and `rem` of struct `div_t`, `ldiv_t` and `lldiv_t` which have the same types.
`lldiv_t lldiv(x,y)`

Multibyte/wide character and string conversions

`mblen(*s, n)` Determines the number of bytes in the multi-byte character pointed to by `s`. At most n characters will be examined. (See also `mbrlen` in [Section 9.1.28](#), [wchar.h](#)).

`mbtowc(*pwc, *s, n)` Converts the multi-byte character in `s` to a wide-character code and stores it in `pwc`. At most n characters will be examined.

`wctomb(*s, wc)` Converts the wide-character `wc` into a multi-byte representation and stores it in the string pointed to by `s`. At most `MB_CUR_MAX` characters are stored.

`mbstowcs(*pwcs, *s, n)` Converts a sequence of multi-byte characters in the string pointed to by `s` into a sequence of wide characters and stores at most n wide characters into the array pointed to by `pwcs`. (See also `mbsrtowcs` in [Section 9.1.28](#), [wchar.h](#)).

`wcstombs(*s, *pwcs, n)` Converts a sequence of wide characters in the array pointed to by `pwcs` into multi-byte characters and stores at most n multi-byte characters into the string pointed to by `s`. (See also `wcsrtowmb` in [Section 9.1.28](#), [wchar.h](#)).

9.1.25. 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

string.h	wchar.h	Description
<code>memcpy(*s1, *s2, n)</code>	<code>wmemcpy(*s1, *s2, n)</code>	Copies <i>n</i> characters from *s2 into *s1 and returns *s1. If *s1 and *s2 overlap the result is undefined.
<code>memmove(*s1, *s2, n)</code>	<code>wmemmove(*s1, *s2, n)</code>	Same as <code>memcpy</code> , but overlapping strings are handled correctly. Returns *s1.
<code>strcpy(*s1, *s2)</code>	<code>wscpy(*s1, *s2)</code>	Copies *s2 into *s1 and returns *s1. If *s1 and *s2 overlap the result is undefined.
<code>strncpy(*s1, *s2, n)</code>	<code>wcsncpy(*s1, *s2, n)</code>	Copies not more than <i>n</i> characters from *s2 into *s1 and returns *s1. If *s1 and *s2 overlap the result is undefined.
<code>strcat(*s1, *s2)</code>	<code>wscat(*s1, *s2)</code>	Appends a copy of *s2 to *s1 and returns *s1. If *s1 and *s2 overlap the result is undefined.
<code>strncat(*s1, *s2, n)</code>	<code>wcsncat(*s1, *s2, n)</code>	Appends not more than <i>n</i> characters from *s2 to *s1 and returns *s1. If *s1 and *s2 overlap the result is undefined.

Comparison functions

string.h	wchar.h	Description
<code>memcmp(*s1, *s2, n)</code>	<code>wmemcmp(*s1, *s2, n)</code>	Compares the first <i>n</i> characters of *s1 to the first <i>n</i> characters of *s2. Returns < 0 if *s1 < *s2, 0 if *s1 == *s2, or > 0 if *s1 > *s2.
<code>strcmp(*s1, *s2)</code>	<code>wscmp(*s1, *s2)</code>	Compares string *s1 to *s2. Returns < 0 if *s1 < *s2, 0 if *s1 == *s2, or > 0 if *s1 > *s2.
<code>strncmp(*s1, *s2, n)</code>	<code>wcsncmp(*s1, *s2, n)</code>	Compares the first <i>n</i> characters of *s1 to the first <i>n</i> characters of *s2. Returns < 0 if *s1 < *s2, 0 if *s1 == *s2, or > 0 if *s1 > *s2.
<code>strcoll(*s1, *s2)</code>	<code>wscoll(*s1, *s2)</code>	Performs a local-specific comparison between string *s1 and string *s2 according to the LC_COLLATE category of the current locale. Returns < 0 if *s1 < *s2, 0 if *s1 == *s2, or > 0 if *s1 > *s2. (See Section 9.1.14, locale.h)
<code>strxfrm(*s1, *s2, n)</code>	<code>wcsxfrm(*s1, *s2, n)</code>	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.

Search functions

string.h	wchar.h	Description
<code>memchr(*s, c, n)</code>	<code>wmemchr(*s, c, n)</code>	Checks the first <i>n</i> characters of *s on the occurrence of character <i>c</i> . Returns a pointer to the found character.
<code>strchr(*s, c)</code>	<code>wchr(*s, c)</code>	Returns a pointer to the first occurrence of character <i>c</i> in *s or the null pointer if not found.

string.h	wchar.h	Description
<code>strrchr(*s,c)</code>	<code>wcsrchr(*s,c)</code>	Returns a pointer to the last occurrence of character <i>c</i> in <i>*s</i> or the null pointer if not found.
<code>strspn(*s,*set)</code>	<code>wcspn(*s,*set)</code>	Searches <i>*s</i> for a sequence of characters specified in <i>*set</i> . Returns the length of the first sequence found.
<code>strcspn(*s,*set)</code>	<code>wcscspn(*s,*set)</code>	Searches <i>*s</i> for a sequence of characters <i>not</i> specified in <i>*set</i> . Returns the length of the first sequence found.
<code>strpbrk(*s,*set)</code>	<code>wcspbrk(*s,*set)</code>	Same as <code>strspn/wcspn</code> but returns a pointer to the first character in <i>*s</i> that also is specified in <i>*set</i> .
<code>strstr(*s,*sub)</code>	<code>wcsstr(*s,*sub)</code>	Searches for a substring <i>*sub</i> in <i>*s</i> . Returns a pointer to the first occurrence of <i>*sub</i> in <i>*s</i> .
<code>strtok(*s,*dlm)</code>	<code>wcstok(*s,*dlm)</code>	A sequence of calls to this function breaks the string <i>*s</i> into a sequence of tokens delimited by a character specified in <i>*dlm</i> . The token found in <i>*s</i> is terminated with a null character. Returns a pointer to the first position in <i>*s</i> of the token.

Miscellaneous functions

string.h	wchar.h	Description
<code>memset(*s,c,n)</code>	<code>wmemset(*s,c,n)</code>	Fills the first <i>n</i> bytes of <i>*s</i> with character <i>c</i> and returns <i>*s</i> .
<code>strerror(errno)</code>	-	Typically, the values for <code>errno</code> come from <code>int errno</code> . This function returns a pointer to the associated error message. (See also Section 9.1.6 , errno.h)
<code>strlen(*s)</code>	<code>wcslength(*s)</code>	Returns the length of string <i>*s</i> .

9.1.26. 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 C99 with one exception: this implementation does not support leap seconds. The `struct tm` type is defines as follows:

```
struct tm
{
    int    tm_sec;        /* seconds after the minute - [0, 59]    */
    int    tm_min;        /* minutes after the hour - [0, 59]      */
    int    tm_hour;       /* hours since midnight - [0, 23]        */
    int    tm_mday;       /* day of the month - [1, 31]            */
    int    tm_mon;        /* months since January - [0, 11]        */
    int    tm_year;       /* year since 1900                        */
    int    tm_wday;       /* days since Sunday - [0, 6]            */
}
```

TASKING VX-toolset for PCP User Guide

```
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 by `CLOCKS_PER_SEC`.

`difftime(t1, t0)` Returns the difference $t1-t0$ in seconds.

`mktime(tm *tp)` Converts the broken-down time in the structure pointed to by *tp*, to a value of type `time_t`. The return value has the same encoding as the return value of the `time` function.

`time(*timer)` Returns the current calendar time. This value is also assigned to *timer*.

Time conversion

`asctime(tm *tp)` Converts the broken-down time in the structure pointed to by *tp* into a string in the form `Mon Jan 22 16:15:14 2007\n\n0`. 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: `asctime(localtime(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`:

<code>time.h</code>	<code>wchar.h</code>
<code>strftime(*<i>s</i>, <i>smax</i>, *<i>fmt</i>, tm *<i>tp</i>)</code>	<code>wstrftime(*<i>s</i>, <i>smax</i>, *<i>fmt</i>, tm *<i>tp</i>)</code>

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 9.1.14, 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 locale-specific date and time representation (same as %a %b %e %T %Y)
 %C last two digits of the year
 %d day of the month (01-31)
 %D same as %m/%d/%y
 %e day of the month (1-31), with single digits preceded by a space
 %F ISO 8601 date format: %Y-%m-%d
 %g last two digits of the week based year (00-99)
 %G week based year (0000–9999)
 %h same as %b
 %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)
 %n replaced by newline character
 %p locale's equivalent of AM or PM
 %r locale's 12-hour clock time; same as %I:%M:%S %p
 %R same as %H:%M
 %S second (00-59)
 %t replaced by horizontal tab character
 %T ISO 8601 time format: %H:%M:%S
 %u ISO 8601 weekday number (1-7), Monday as first day of the week
 %U week number of the year (00-53), week 1 has the first Sunday
 %V ISO 8601 week number (01-53) in the week-based year
 %w weekday (0-6, Sunday is 0)
 %W week number of the year (00-53), week 1 has the first Monday
 %x local date representation
 %X local time representation
 %y year without century (00-99)
 %Y year with century
 %z ISO 8601 offset of time zone from UTC, or nothing
 %Z time zone name, if any
 %% %

9.1.27. unistd.h

The file `unistd.h` contains standard UNIX I/O functions. These functions are all implemented using file system simulation. Except for `lstat` and `fstat` which are not implemented. This header file is not defined in ISO C99.

`access(*name, mode)` Use file system simulation 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 file system simulation 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 file system simulation 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 file system simulation to `stat()` a file on the host platform. *(FSS implementation)*

`lstat(*name, *buff)` This function is identical to `stat()`, except in the case of a symbolic link, where the link itself is 'stat'-ed, not the file that it refers to. *(Not implemented)*

`fstat(fd, *buff)` This function is identical to `stat()`, except that it uses a file descriptor instead of a name. *(Not implemented)*

`unlink(*name)` Removes the named file, so that a subsequent attempt to open it fails. *(FSS implementation)*

`write(fd, *buff, cnt)` Write a sequence of characters to a file. Calls `_write()`. *(FSS implementation)*

9.1.28. wchar.h

Many functions in `wchar.h` represent the wide-character variant of other functions so these are discussed together. (See [Section 9.1.23, `stdio.h` and `wchar.h`](#), [Section 9.1.24, `stdlib.h` and `wchar.h`](#), [Section 9.1.25, `string.h` and `wchar.h`](#) and [Section 9.1.26, `time.h` and `wchar.h`](#)).

The remaining functions are described below. They perform conversions between multi-byte characters and wide characters. In these functions, *ps* points to struct `mbstate_t` which holds the conversion state information necessary to convert between sequences of multibyte characters and wide characters:

```
typedef struct
{
```



```

wchar_t      wc_value; /* wide character value solved
                        so far */
unsigned short n_bytes; /* number of bytes of solved
                        multibyte */
unsigned short encoding; /* encoding rule for wide
                        character <=> multibyte
                        conversion */
} mbstate_t;

```

When multibyte characters larger than 1 byte are used, this struct will be used to store the conversion information when not all the bytes of a particular multibyte character have been read from the source. In this implementation, multi-byte characters are 1 byte long (MB_CUR_MAX and MB_LEN_MAX are defined as 1) and this will never occur.

<code>mbsinit(*ps)</code>	Determines whether the object pointed to by <i>ps</i> , is an initial conversion state. Returns a non-zero value if so.
<code>mbsrtowcs(*pwcs, **src, n, *ps)</code>	Restartable version of <code>mbstowcs</code> . See Section 9.1.24, <code>stdlib.h</code> and <code>wchar.h</code> . The initial conversion state is specified by <i>ps</i> . The input sequence of multibyte characters is specified indirectly by <i>src</i> .
<code>wcsrtombs(*s, **src, n, *ps)</code>	Restartable version of <code>wcstombs</code> . See Section 9.1.24, <code>stdlib.h</code> and <code>wchar.h</code> . The initial conversion state is specified by <i>ps</i> . The input wide string is specified indirectly by <i>src</i> .
<code>mbrtowc(*pwc, *s, n, *ps)</code>	Converts a multibyte character <i>s</i> to a wide character <i>pwc</i> according to conversion state <i>ps</i> . See also <code>mbtowc</code> in Section 9.1.24, <code>stdlib.h</code> and <code>wchar.h</code> .
<code>wcrtomb(*s, wc, *ps)</code>	Converts a wide character <i>wc</i> to a multi-byte character according to conversion state <i>ps</i> and stores the multi-byte character in <i>s</i> .
<code>btowc(c)</code>	Returns the wide character corresponding to character <i>c</i> . Returns WEOF on error.
<code>wctob(c)</code>	Returns the multi-byte character corresponding to the wide character <i>c</i> . The returned multi-byte character is represented as one byte. Returns EOF on error.
<code>mbrlen(*s, n, *ps)</code>	Inspects up to <i>n</i> bytes from the string <i>s</i> to see if those characters represent valid multibyte characters, relative to the conversion state held in <i>ps</i> .

9.1.29. `wctype.h`

Most functions in `wctype.h` represent the wide-character variant of functions declared in `ctype.h` and are discussed in [Section 9.1.4, `ctype.h` and `wctype.h`](#). In addition, this header file provides extensible, locale specific functions and wide character classification.

<code>wctype(*property)</code>	Constructs a value of type <code>wctype_t</code> that describes a class of wide characters identified by the string <i>property</i> . If <i>property</i> identifies a valid class of wide characters according to the LC_TYPE category (see Section 9.1.14, <code>locale.h</code>) of the current locale, a non-zero value is returned that can be used as an argument in the <code>iswctype</code> function.
--------------------------------	--

TASKING VX-toolset for PCP User Guide

`iswctype(wc, desc)` Tests whether the wide character `wc` is a member of the class represented by `wctype_t desc`. Returns a non-zero value if tested true.

Function	Equivalent to locale specific test
<code>iswalnum(wc)</code>	<code>iswctype(wc, wctype("alnum"))</code>
<code>iswalpha(wc)</code>	<code>iswctype(wc, wctype("alpha"))</code>
<code>iswcntrl(wc)</code>	<code>iswctype(wc, wctype("cntrl"))</code>
<code>iswdigit(wc)</code>	<code>iswctype(wc, wctype("digit"))</code>
<code>iswgraph(wc)</code>	<code>iswctype(wc, wctype("graph"))</code>
<code>iswlower(wc)</code>	<code>iswctype(wc, wctype("lower"))</code>
<code>iswprint(wc)</code>	<code>iswctype(wc, wctype("print"))</code>
<code>iswpunct(wc)</code>	<code>iswctype(wc, wctype("punct"))</code>
<code>iswspace(wc)</code>	<code>iswctype(wc, wctype("space"))</code>
<code>iswupper(wc)</code>	<code>iswctype(wc, wctype("upper"))</code>
<code>iswxdigit(wc)</code>	<code>iswctype(wc, wctype("xdigit"))</code>

`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 [Section 9.1.14, locale.h](#)) of the current locale, a non-zero value is returned that can be used as an argument in the `towctrans` function.

`towctrans(wc, desc)` Transforms wide character `wc` into another wide-character, described by `desc`.

Function	Equivalent to locale specific transformation
<code>towlower(wc)</code>	<code>towctrans(wc, wctrans("tolower"))</code>
<code>toupper(wc)</code>	<code>towctrans(wc, wctrans("toupper"))</code>

9.2. C Library Reentrancy

Some of the functions in the C library are reentrant, others are not. The table below shows the functions in the C library, and whether they are reentrant or not. A dash means that the function is reentrant. Note that some of the functions are not reentrant because they set the global variable 'errno' (or call other functions that eventually set 'errno'). If your program does not check this variable and `errno` is the only reason for the function not being reentrant, these functions can be assumed reentrant as well.

The explanation of the cause why a function is not reentrant sometimes refers to a footnote because the explanation is too lengthy for the table.

Function	Not reentrant because
<code>_close</code>	Uses global File System Simulation buffer, <code>_dbg_request</code>
<code>_doflt</code>	Uses I/O functions which modify <code>job[]</code> . See (1).
<code>_doprint</code>	Uses indirect access to static <code>job[]</code> array. See (1).

Function	Not reentrant because
<code>_doscan</code>	Uses indirect access to <code>iob[]</code> and calls <code>ungetc</code> (access to local static <code>ungetc[]</code> buffer). See (1).
<code>_Exit</code>	See <code>exit</code> .
<code>_filbuf</code>	Uses <code>iob[]</code> , which is not reentrant. See (1).
<code>_flsbuf</code>	Uses <code>iob[]</code> . See (1).
<code>_getflt</code>	Uses <code>iob[]</code> . See (1).
<code>_iob</code>	Defines static <code>iob[]</code> . See (1).
<code>_lseek</code>	Uses global File System Simulation buffer, <code>_dbg_request</code>
<code>_open</code>	Uses global File System Simulation buffer, <code>_dbg_request</code>
<code>_read</code>	Uses global File System Simulation buffer, <code>_dbg_request</code>
<code>_unlink</code>	Uses global File System Simulation buffer, <code>_dbg_request</code>
<code>_write</code>	Uses global File System Simulation buffer, <code>_dbg_request</code>
<code>abort</code>	Calls <code>exit</code>
<code>abs labs llabs</code>	-
<code>access</code>	Uses global File System Simulation buffer, <code>_dbg_request</code>
<code>acos acosf acosl</code>	Sets <code>errno</code> .
<code>acosh acoshf acoshl</code>	Sets <code>errno</code> via calls to other functions.
<code>asctime</code>	<code>asctime</code> defines static array for broken-down time string.
<code>asin asinf asinl</code>	Sets <code>errno</code> .
<code>asinh asinhf asinhl</code>	Sets <code>errno</code> via calls to other functions.
<code>atan atanf atanl</code>	-
<code>atan2 atan2f atan2l</code>	-
<code>atanh atanhf atanhl</code>	Sets <code>errno</code> via calls to other functions.
<code>atexit</code>	<code>atexit</code> defines static array with function pointers to execute at exit of program.
<code>atof</code>	-
<code>atoi</code>	-
<code>atol</code>	-
<code>bsearch</code>	-
<code>btowc</code>	-
<code>cabs cabsf cabsl</code>	Sets <code>errno</code> via calls to other functions.
<code>cacos cacosf cacosl</code>	Sets <code>errno</code> via calls to other functions.
<code>cacosh cacosh cfacoshl</code>	Sets <code>errno</code> via calls to other functions.
<code>calloc</code>	<code>calloc</code> uses static buffer management structures. See <code>malloc</code> (5).
<code>carg cargf cargl</code>	-
<code>casin casinf casinl</code>	Sets <code>errno</code> via calls to other functions.

Function	Not reentrant because
casinh casinh cfasinhl	Sets errno via calls to other functions.
catan catanf catanl	Sets errno via calls to other functions.
catanh catanhf catanhl	Sets errno via calls to other functions.
cbirt cbirtf cbirtl	<i>(Not implemented)</i>
ccos ccoshf ccosl	Sets errno via calls to other functions.
ccosh ccoshf ccoshl	Sets errno via calls to other functions.
ceil ceilf ceill	-
cexp cexpf cexpl	Sets errno via calls to other functions.
chdir	Uses global File System Simulation buffer, <code>_dbg_request</code>
cimag cimagf cimagl	-
cleanup	Calls <code>fclose</code> . See (1)
clearerr	Modifies <code>io[]</code> . See (1)
clock	Uses global File System Simulation buffer, <code>_dbg_request</code>
clog clogf clogl	Sets errno via calls to other functions.
close	Calls <code>_close</code>
conj conjf conjl	-
copysign copysignf copysignl	-
cos cosf cosl	-
cosh coshf coshl	<code>cosh</code> calls <code>exp()</code> , which sets <code>errno</code> . If <code>errno</code> is discarded, <code>cosh</code> is reentrant.
cpow cpowf cpowl	Sets errno via calls to other functions.
cproj cprojf cprojl	-
creal crealf creall	-
csin csinf csinl	Sets errno via calls to other functions.
csinh csinhf csinhl	Sets errno via calls to other functions.
csqrt csqrtf csqrtl	Sets errno via calls to other functions.
ctan ctanf ctanl	Sets errno via calls to other functions.
ctanh ctanhf ctanhl	Sets errno via calls to other functions.
ctime	Calls <code>asctime</code>
difftime	-
div ldiv lldiv	-
erf erfl erff	<i>(Not implemented)</i>
erfc erfcl erfcl	<i>(Not implemented)</i>
exit	Calls <code>fclose</code> indirectly which uses <code>io[]</code> calls functions in <code>_atexit</code> array. See (1). To make <code>exit</code> reentrant kernel support is required.

Function	Not reentrant because
exp expf expl	Sets errno.
exp2 exp2f exp2l	<i>(Not implemented)</i>
expm1 expm1f expm1l	<i>(Not implemented)</i>
fabs fabsf fabsl	-
fclose	Uses values in iob[]. See (1).
fdim fdimf fdiml	<i>(Not implemented)</i>
feclearexcept	<i>(Not implemented)</i>
fegetenv	<i>(Not implemented)</i>
fegetexceptflag	<i>(Not implemented)</i>
fegetround	<i>(Not implemented)</i>
feholdexcept	<i>(Not implemented)</i>
feof	Uses values in iob[]. See (1).
feraiseexcept	<i>(Not implemented)</i>
ferror	Uses values in iob[]. See (1).
fesetenv	<i>(Not implemented)</i>
fesetexceptflag	<i>(Not implemented)</i>
fesetround	<i>(Not implemented)</i>
fetestexcept	<i>(Not implemented)</i>
feupdateenv	<i>(Not implemented)</i>
fflush	Modifies iob[]. See (1).
fgetc fgetwc	Uses pointer to iob[]. See (1).
fgetpos	Sets the variable errno and uses pointer to iob[]. See (1) / (2).
fgets fgetws	Uses iob[]. See (1).
floor floorf floorl	-
fma fmaf fmal	<i>(Not implemented)</i>
fmax fmaxf fmaxl	<i>(Not implemented)</i>
fmin fminf fminl	<i>(Not implemented)</i>
fmod fmodf fmodl	-
fopen	Uses iob[] and calls malloc when file open for buffered IO. See (1)
fpclassify	-
fprintf fwprintf	Uses iob[]. See (1).
fputc fputwc	Uses iob[]. See (1).
fputs fputws	Uses iob[]. See (1).
fread	Calls fgetc. See (1).
free	free uses static buffer management structures. See malloc (5).

Function	Not reentrant because
freopen	Modifies iob[]. See (1).
frexp frexpf frexpl	-
fscanf fwscanf	Uses iob[]. See (1)
fseek	Uses iob[] and calls _lseek. Accesses ungetc[] array. See (1).
fsetpos	Uses iob[] and sets errno. See (1) / (2).
fstat	<i>(Not implemented)</i>
ftell	Uses iob[] and sets errno. Calls _lseek. See (1) / (2).
fwrite	Uses iob[]. See (1).
getc getwc	Uses iob[]. See (1).
getchar getwchar	Uses iob[]. See (1).
getcwd	Uses global File System Simulation buffer, _dbg_request
getenv	Skeleton only.
gets getws	Uses iob[]. See (1).
gmtime	gmtime defines static structure
hypot hypotf hypotl	Sets errno via calls to other functions.
ilogb ilogbf ilogbl	<i>(Not implemented)</i>
imaxabs	-
imaxdiv	-
isalnum iswalnum	-
isalpha iswalpha	-
isascii iswascii	-
iscntrl iswcntrl	-
isdigit iswdigit	-
isfinite	-
isgraph iswgraph	-
isgreater	-
isgreaterequal	-
isinf	-
isless	-
islessequal	-
islessgreater	-
islower iswlower	-
isnan	-
isnormal	-
isprint iswprint	-

Function	Not reentrant because
ispunct iswpunct	-
isspace iswspace	-
isunordered	-
isupper iswupper	-
iswalnum	-
iswalpha	-
iswcntrl	-
iswctype	-
iswdigit	-
iswgraph	-
iswlower	-
iswprint	-
iswpunct	-
iswspace	-
iswupper	-
iswxditig	-
isxdigit iswxdigit	-
ldexp ldexpf ldexpl	Sets errno. See (2).
lgamma lgammaf lgammal	<i>(Not implemented)</i>
llrint lrintf lrintl	<i>(Not implemented)</i>
llround llroundf llroundl	<i>(Not implemented)</i>
localeconv	N.A.; skeleton function
localtime	-
log logf logl	Sets errno. See (2).
log10 log10f log10l	Sets errno via calls to other functions.
log1p log1pf log1pl	<i>(Not implemented)</i>
log2 log2f log2l	<i>(Not implemented)</i>
logb logbf logbl	<i>(Not implemented)</i>
longjmp	-
lrint lrintf lrintl	<i>(Not implemented)</i>
lround lroundf lroundl	<i>(Not implemented)</i>
lseek	Calls _lseek
lstat	<i>(Not implemented)</i>
malloc	Needs kernel support. See (5).
mblen	N.A., skeleton function

Function	Not reentrant because
mbrlen	Sets errno.
mbrtowc	Sets errno.
mbsinit	-
mbsrtowcs	Sets errno.
mbstowcs	N.A., skeleton function
mbtowc	N.A., skeleton function
memchr wmemchr	-
memcmp wmemcmp	-
memcpy wmemcpy	-
memmove wmemmove	-
memset wmemset	-
mktime	-
modf modff modfl	-
nan nanf nanl	<i>(Not implemented)</i>
nearbyint nearbyintf nearbyintl	<i>(Not implemented)</i>
nextafter nextafterf nextafterl	<i>(Not implemented)</i>
nexttoward nexttowardf nexttowardl	<i>(Not implemented)</i>
offsetof	-
open	Calls <code>_open</code>
perror	Uses errno. See (2)
pow powf powl	Sets errno. See (2)
printf wprintf	Uses <code>iob[]</code> . See (1)
putc putwc	Uses <code>iob[]</code> . See (1)
putchar putwchar	Uses <code>iob[]</code> . See (1)
puts	Uses <code>iob[]</code> . See (1)
qsort	-
raise	Updates the signal handler table
rand	Uses static variable to remember latest random number. Must diverge from ISO C standard to define reentrant rand. See (4).
read	Calls <code>_read</code>
realloc	See malloc (5).
remainder remainderf remainderl	<i>(Not implemented)</i>

Function	Not reentrant because
remove	Uses global File System Simulation buffer, <code>_dbg_request</code>
remquo remquof remquol	(Not implemented)
rename	Uses global File System Simulation buffer, <code>_dbg_request</code>
rewind	Eventually calls <code>_lseek</code>
rint rintf rintl	(Not implemented)
round roundf roundl	(Not implemented)
scalbln scalblnf scalblnl	-
scalbn scalbnf scalbnl	-
scanf wscanf	Uses <code>job[]</code> , calls <code>_doscan</code> . See (1).
setbuf	Sets <code>job[]</code> . See (1).
setjmp	-
setlocale	N.A.; skeleton function
setvbuf	Sets <code>job</code> and calls <code>malloc</code> . See (1) / (5).
signal	Updates the signal handler table
signbit	-
sin sinf sinl	-
sinh sinhf sinhl	Sets <code>errno</code> via calls to other functions.
snprintf swprintf	Sets <code>errno</code> . See (2).
sprintf	Sets <code>errno</code> . See (2).
sqrt sqrtf sqrtl	Sets <code>errno</code> . See (2).
srand	See <code>rand</code>
sscanf swscanf	Sets <code>errno</code> via calls to other functions.
stat	Uses global File System Simulation buffer, <code>_dbg_request</code>
strcat wscat	-
strchr wcschr	-
strcmp wcscmp	-
strcoll wscoll	-
strcpy wcsncpy	-
strcspn wcsncpy	-
strerror	-
strftime wstrftime	-
strlen wcslen	-
strncat wcsncat	-
strncmp wcsncmp	-
strncpy wcsncpy	-

Function	Not reentrant because
strpbrk wcsprk	-
strrchr wcsrchr	-
strspn wcsspn	-
strstr wcsstr	-
strtod wctod	-
strtod wctod	-
strtoimax	Sets errno via calls to other functions.
strtok wctok	strtok saves last position in string in local static variable. This function is not reentrant by design. See (4).
strtol wcstol	Sets errno. See (2).
strtold wcstold	-
strtoul wcstoul	Sets errno. See (2).
strtoull wcstoull	Sets errno. See (2).
strtoumax	Sets errno via calls to other functions.
strxfrm wcsxfrm	-
system	N.A; skeleton function
tan tanf tanl	Sets errno. See (2).
tanh tanhf tanhl	Sets errno via call to other functions.
tgamma tgammaf tgamma1	<i>(Not implemented)</i>
time	Uses static variable which defines initial start time
tmpfile	Uses iob[]. See (1).
tmpnam	Uses local buffer to build filename. Function can be adapted to use user buffer. This makes the function non ISO C. See (4).
toascii	-
tolower	-
toupper	-
towctrans	-
towlower	-
towupper	-
trunc truncf trunc1	<i>(Not implemented)</i>
ungetc ungetwc	Uses static buffer to hold ungetc characters for each file. Can be moved into iob structure. See (1).
unlink	Uses global File System Simulation buffer, _dbg_request
vfprintf vfwprintf	Uses iob[]. See (1).
vscanf vfwscanf	Calls _doscan

Function	Not reentrant because
<code>vprintf</code> <code>vwprintf</code>	Uses <code>job[]</code> . See (1).
<code>vscanf</code> <code>vwscanf</code>	Calls <code>_doscan</code>
<code>vsprintf</code> <code>vswprintf</code>	Sets <code>errno</code> .
<code>vsscanf</code> <code>vswscanf</code>	Sets <code>errno</code> .
<code>wcrtomb</code>	Sets <code>errno</code> .
<code>wcsrtombs</code>	Sets <code>errno</code> .
<code>wcstoimax</code>	Sets <code>errno</code> via calls to other functions.
<code>wcstombs</code>	N.A.; skeleton function
<code>wcstoumax</code>	Sets <code>errno</code> via calls to other functions.
<code>wctob</code>	-
<code>wctomb</code>	N.A.; skeleton function
<code>wctrans</code>	-
<code>wctype</code>	-
<code>write</code>	Calls <code>_write</code>

Table: C library reentrancy

Several functions in the C library are not reentrant due to the following reasons:

- The `job[]` structure is static. This influences all I/O functions.
- The `ungetc[]` array is static. This array holds the characters (one for each stream) when `ungetc()` is called.
- The variable `errno` is globally defined. Numerous functions read or modify `errno`
- `_doprint` and `_doscan` use static variables for e.g. character counting in strings.
- Some string functions use locally defined (static) buffers. This is prescribed by ANSI.
- `malloc` uses a static heap space.

The following description discusses these items into more detail. The numbers at the begin of each paragraph relate to the number references in the table above.

(1) *job* structures

The I/O part of the C library is not reentrant by design. This is mainly caused by the static declaration of the `job[]` array. The functions which use elements of this array access these elements via pointers (`FILE *`).

Building a multi-process system that is created in one link-run is hard to do. The C language scoping rules for external variables make it difficult to create a private copy of the `job[]` array. Currently, the `job[]` array has external scope. Thus it is visible in every module involved in one link phase. If these modules comprise several tasks (processes) in a system each of which should have its private copy of

TASKING VX-toolset for PCP User Guide

`iob[]`, it is apparent that the `iob[]` declaration should be changed. This requires adaptation of the library to the multi-tasking environment. The library modules must use a process identification as an index for determining which `iob[]` array to use. Thus the library is suitable for interfacing to that kernel only.

Another approach for the `iob[]` declaration problem is to declare the array static in one of the modules which create a task. Thus there can be more than one `iob[]` array in the system without having conflicts at link time. This brings several restrictions: Only the module that holds the declaration of the static `iob[]` can use the standard file handles `stdin`, `stdout` and `stderr` (which are the first three entries in `iob[]`). Thus all I/O for these three file handles should be located in one module.

(2) *errno* declaration

Several functions in the C library set the global variable `errno`. After completion of the function the user program may consult this variable to see if some error occurred. Since most of the functions that set `errno` already have a return type (this is the reason for using `errno`) it is not possible to check successful completion via the return type.

The library routines can set `errno` to the values defined in `errno.h`. See the file `errno.h` for more information.

`errno` can be set to `ERR_FORMAT` by the print and scan functions in the C library if you specify illegal format strings.

`errno` will never be set to `ERR_NOLONG` or `ERR_NOPOINT` since the C library supports long and pointer conversion routines for input and output.

`errno` can be set to `ERANGE` by the following functions: `exp()`, `strtol()`, `strtoul()` and `tan()`. These functions may produce results that are out of the valid range for the return type. If so, the result of the function will be the largest representable value for that type and `errno` is set to `ERANGE`.

`errno` is set to `EDOM` by the following functions: `acos()`, `asin()`, `log()`, `pow()` and `sqrt()`. If the arguments for these functions are out of their valid range (e.g. `sqrt(-1)`), `errno` is set to `EDOM`.

`errno` can be set to `ERR_POS` by the file positioning functions `ftell()`, `fsetpos()` and `fgetpos()`.

(3) *ungetc*

Currently the `ungetc` buffer is static. For each file entry in the `iob[]` structure array, there is one character available in the buffer to `ungetc` a character.

(4) *local buffers*

`tmpnam()` creates a temporary filename and returns a pointer to a local static buffer. This is according to the ANSI definition. Changing this function such that it creates the name in a user specified buffer requires another calling interface. Thus the function would be no longer portable.

`strtok()` scans through a string and remembers that the string and the position in the string for subsequent calls. This function is not reentrant by design. Making it reentrant requires support of a kernel to store the information on a per process basis.

`rand()` generates a sequence of random numbers. The function uses the value returned by a previous call to generate the next value in the sequence. This function can be made reentrant by specifying the previous random value as one of the arguments. However, then it is no longer a standard function.

(5) malloc

Malloc uses a heap space which is assigned at locate time. Thus this implementation is not reentrant. Making a reentrant malloc requires some sort of system call to obtain free memory space on a per process basis. This is not easy to solve within the current context of the library. This requires adaptation to a kernel.

This paragraph on reentrancy applies to multi-process environments only. If reentrancy is required for calling library functions from an exception handler, another approach is required. For such a situation it is of no use to allocate e.g. multiple `iob[]` structures. In such a situation several pieces of code in the library have to be declared 'atomic': this means that interrupts have to be disabled while executing an atomic piece of code.

Chapter 10. List File Formats

This chapter describes the format of the assembler list file and the linker map file.

10.1. Assembler List File Format

The assembler list file is an additional output file of the assembler that contains information about the generated code. For details on how to generate a list file, see [Section 4.5, *Generating a List File*](#).

The list file consists of a page header and a source listing.

Page header

The page header is repeated on every page:

```
TASKING VX-toolset for PCP: PCP assembler vx.yrz Build nnn SN 00000000
Title                                                                    Page 1
```

```
ADDR CODE      CYCLES  LINE SOURCE LINE
```

The first line contains version information. The second line can contain a title which you can specify with the assembler control `$TITLE` and always contains a page number. The third line is empty and the fourth line contains the headings of the columns for the source listing.

With the assembler controls `$LIST`, `$PAGE`, and with the [assembler option `--list-format`](#) you can format the list file.

Source listing

The following is a sample part of a listing. An explanation of the different columns follows below.

```
ADDR CODE      CYCLES  LINE SOURCE LINE
                1          ; Module start
                .
                .
0003 93C0rrrr          27      ld.lil  r7,@DPTR(_PCP_world)
0005 53rr            28      ld.pi   r5,[_PCP_world]
0006 93C0rrrr          29      ld.lil  r7,@DPTR(_PCP_data_printf)
0008 55rr            30      st.pi   r5,[_PCP_data_printf]
                .
                .
0000                44 buf:   .space  4
  | RESERVED
0003
```

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.

TASKING VX-toolset for PCP User Guide

CODE	This is the object code generated by the assembler for this source line, displayed in hexadecimal format. The displayed code need not be the same as the generated code that is entered in the object module. The code can also be relocatable code. In this case the letter 'r' is printed for the relocatable code part in the listing. For lines that allocate space, the code field contains the text "RESERVED". For lines that initialize a buffer, the code field lists one value followed by the word "REPEATS".
CYCLES	The first number in this column is the number of instruction cycles needed to execute the instruction(s) as generated in the CODE field. The second number is the accumulated cycle count of this section.
LINE	This column contains the line number. This is a decimal number indicating each input line, starting from 1 and incrementing with each source line.
SOURCE LINE	This column contains the source text. This is a copy of the source line from the assembly source file.

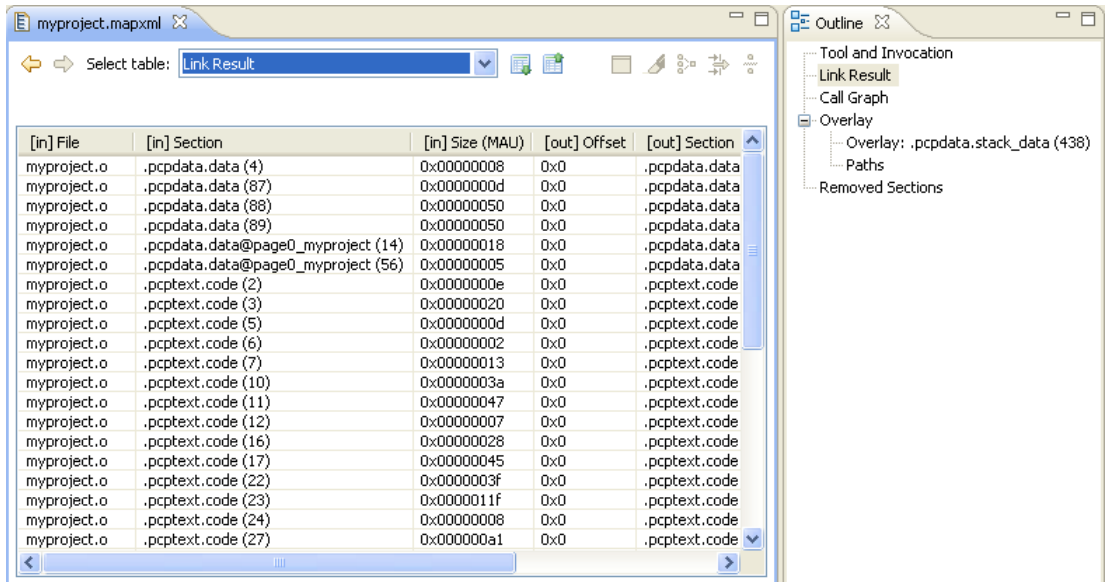
For the `.SET` and `.EQU` directives the `ADDR` and `CODE` columns do not apply. The symbol value is listed instead.

10.2. Linker Map File Format

The linker map file is an additional output file of the linker that shows how the linker has mapped the sections and symbols from the various object files (`.o`) to output sections. Locate information is not present, because that is not available for a PCP project. External symbols are listed per space with their absolute address, both sorted on symbol and sorted on address. For details on how to generate a map file, see [Section 5.9, *Generating a Map File*](#).

With the linker option `--map-file-format` you can specify which parts of the map file you want to see.

In Eclipse the linker map file (`project.map.xml`) is generated in the output directory of the build configuration, usually `Debug` or `Release`. You can open the map file by double-clicking on the file name.



Each page displays a part of the map file. You can use the drop-down list or the Outline view to navigate through the different tables and you can use the following buttons.

Icon	Action	Description
	Back	Goes back one page in the history list.
	Forward	Goes forward one page in the history list.
	Next Table	Shows the next table from the drop-down list.
	Previous Table	Shows the previous table from the drop-down list.

When you right-click in the view, a popup menu appears (for example, to reset the layout of a table). The meaning of the different parts is:

Tool and Invocation

This part of the map file contains information about the linker, its version header information, binary location and which options are used to call it.

Processed Files

This part of the map file shows all processed files. This also includes object files that are extracted from a library, with the symbol that led to the extraction. This part is not available when you use MIL linking (control program option `--mil-link`).

Link Result

This part of the map file shows per object file how the link phase has mapped the sections from the various object files (. o) to output sections.

[in] File	The name of an input object file.
[in] Section	A section name and id from the input object file. The number between '(')' uniquely identifies the section.
[in] Size	The size of the input section.
[out] Offset	The offset relative to the start of the output section.
[out] Section	The resulting output section name and id.
[out] Size	The size of the output section.

Module Local Symbols

This part of the map file shows a table for each local scope within an object file. Each table has three columns, 1 the symbol name, 2 the address of the symbol and 3 the space where the symbol resides in. The table is sorted on symbol name within each space.



By default this part is not shown in the map file. You have to turn this part on manually with [linker option --map-file-format=+statics](#) (module local symbols).





Cross References

This part of the map file lists all symbols defined in the object modules and for each symbol the object modules that contain a reference to the symbol are shown. Also, symbols that remain undefined are shown. This part is not available when you use MIL linking ([control program option --mil-link](#)).

Call Graph

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.

You can click the + or - sign to expand or collapse a single node. Use the  /  buttons to expand/collapse all nodes in the call graph.

Icon	Meaning	Description
	Root	This function is the top of the call graph. If there are interrupt handlers, there can be several roots.
	Callee	This function is referenced by several No leaf functions. Right-click on the function and select Expand all References to see all functions that reference this function. Select Back to Caller to return to the calling function.
	Node	A normal node (function) in the call graph.
	Caller	This function calls a function which is listed separately in the call graph. Right-click on the function and select Go to Callee to see the callee. Hover the mouse over the function to see a popup with all callees.

Overlay

This part of the map file shows how the stack is organized. This part also shows the locate overlay information if you used overlay groups in the linker script file.

Processor and Memory

This part of the map file shows the processor and memory information of the linker script file.

By default this part is not shown in the map file. You have to turn this part on manually with [linker option `--map-file-format=+lsl`](#) (processor and memory info). You can print this information to a separate file with [linker option `--lsl-dump`](#).

You can click the + or - sign to expand or collapse a part of the information.

Removed Sections

This part of the map file shows the sections which are removed from the output file as a result of the optimization option to delete unreferenced sections and or duplicate code or constant data ([linker option `--optimize=cxy`](#)).

Section	The name of the section which has been removed.
File	The name of the input object file where the section is removed from.
Library	The name of the library where the object file is part of.
Symbol	The symbols that were present in the section.
Reason	The reason why the section has been removed. This can be because the section is unreferenced or duplicated.

Chapter 11. Linker Script Language (LSL)

To make full use of the linker, you can write a script with information about the architecture of the target processor and locating information. The language for the script is called the *Linker Script Language (LSL)*. This chapter first describes the structure of an LSL file. The next section contains a summary of the LSL syntax. In the remaining sections, the semantics of the Linker Script Language is explained.

The TASKING linker is a target independent linker/locator that can simultaneously link and locate all programs for all cores available on a target board. The target board may be of arbitrary complexity. A simple target board may contain one standard processor with some external memory that executes one task. A complex target board may contain multiple standard processors and DSPs combined with configurable IP-cores loaded in an FPGA. Each core may execute a different program, and external memory may be shared by multiple cores.

LSL serves two purposes. First it enables you to specify the characteristics (that are of interest to the linker) of your specific target board and of the cores installed on the board. Second it enables you to specify how sections should be located in memory.

11.1. Structure of a Linker Script File

A script file consists of several definitions. The definitions can appear in any order.

The architecture definition (required)

In essence an *architecture definition* describes how the linker should convert logical addresses into physical addresses for a given type of core. If the core supports multiple address spaces, then for each space the linker must know how to perform this conversion. In this context a physical address is an offset on a given internal or external bus. Additionally the architecture definition contains information about items such as the (hardware) stack and the interrupt vector table.

This specification is normally written by Altium. Altium supplies LSL files in the `include.lsl` directory. The architecture definition of the LSL file should not be changed by you unless you also modify the core's hardware architecture. If the LSL file describes a multi-core system an architecture definition must be available for each different type of core.

See [Section 11.4, *Semantics of the Architecture Definition*](#) for detailed descriptions of LSL in the architecture definition.

The derivative definition

The *derivative definition* describes the configuration of the internal (on-chip) bus and memory system. Basically it tells the linker how to convert offsets on the buses specified in the architecture definition into offsets in internal memory. Microcontrollers and DSPs often have internal memory and I/O sub-systems apart from one or more cores. The design of such a chip is called a *derivative*.

Altium provides LSL descriptions of supported derivatives, 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 11.5, *Semantics of the Derivative Definition*](#) for a detailed description of LSL in the derivative definition.

The processor definition

The *processor definition* describes an instance of a derivative. Typically the processor definition instantiates one derivative only (single-core processor). A processor that contains multiple cores having the same (homogeneous) or different (heterogeneous) architecture can also be described by instantiating multiple derivatives of the same or different types in separate processor definitions.

See [Section 11.6, *Semantics of the Board Specification*](#) for a detailed description of LSL in the processor definition.

The memory and bus definitions (optional)

Memory and bus definitions are used within the context of a derivative definition to specify internal memory and on-chip buses. In the context of a board specification the memory and bus definitions are used to define external (off-chip) memory and buses. Given the above definitions the linker can convert a logical address into an offset into an on-chip or off-chip memory device.

See [Section 11.6.3, *Defining External Memory and Buses*](#), for more information on how to specify the external physical memory layout. *Internal* memory for a processor should be defined in the derivative definition for that processor.

The board specification

The processor definition and memory and bus definitions together form a *board specification*. LSL provides language constructs to easily describe single-core and heterogeneous or homogeneous multi-core systems. The board specification describes all characteristics of your target board's system buses, memory devices, I/O sub-systems, and cores that are of interest to the linker. Based on the information provided in the board specification the linker can for each core:

- convert a logical address to an offset within a memory device
- locate sections in physical memory
- maintain an overall view of the used and free physical memory within the whole system while locating

The section layout definition (optional)

The optional section layout definition enables you to exactly control where input sections are located. Features are provided such as: the ability to place sections at a given load-address or run-time address, to place sections in a given order, and to overlay code and/or data sections.

Which object files (sections) constitute the task that will run on a given core is specified on the command line when you invoke the linker. The linker will link and locate all sections of all tasks simultaneously. From the section layout definition the linker can deduce where a given section may be located in memory,

from the board specification the linker can deduce which physical memory is (still) available while locating the section.

See [Section 11.8, *Semantics of the Section Layout Definition*](#), for more information on how to locate a section at a specific place in memory.

Skeleton of a Linker Script File

```
architecture architecture_name
{
    // Specification core architecture
}

derivative derivative_name
{
    // Derivative definition
}

processor processor_name
{
    // Processor definition
}

memory and/or bus definitions

section_layout space_name
{
    // section placement statements
}
```

11.2. Syntax of the Linker Script Language

This section describes what the LSL language looks like. An LSL document is stored as a file coded in UTF-8 with extension `.lsl`. Before processing an LSL file, the linker preprocesses it using a standard C preprocessor. Following this, the linker interprets the LSL file using a scanner and parser. Finally, the linker uses the information found in the LSL file to guide the locating process.

11.2.1. 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:

```
#include "arch.lsl"
```

Preprocess and include the file `arch.lsl` at this point in the LSL file.

11.2.2. Lexical Syntax

The following lexicon is used to describe the syntax of the Linker Script Language:

$A ::= B$	=	A is defined as B
$A ::= B C$	=	A is defined as B and C ; B is followed by C
$A ::= B \mid C$	=	A is defined as B or C
$\langle B \rangle^{0 1}$	=	zero or one occurrence of B
$\langle B \rangle^{>=0}$	=	zero or more occurrences of B
$\langle B \rangle^{>=1}$	=	one or more occurrences of B
<i>IDENTIFIER</i>	=	a character sequence starting with 'a'-'z', 'A'-'Z' or '_'. Following characters may also be digits and dots '.'
<i>STRING</i>	=	sequence of characters not starting with \n, \r or \t
<i>DQSTRING</i>	=	" <i>STRING</i> " (double quoted string)
<i>OCT_NUM</i>	=	octal number, starting with a zero (06, 045)
<i>DEC_NUM</i>	=	decimal number, not starting with a zero (14, 1024)
<i>HEX_NUM</i>	=	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 `/**/`.

11.2.3. Identifiers and Tags

<i>arch_name</i>	::=	<i>IDENTIFIER</i>
<i>bus_name</i>	::=	<i>IDENTIFIER</i>
<i>core_name</i>	::=	<i>IDENTIFIER</i>
<i>derivative_name</i>	::=	<i>IDENTIFIER</i>
<i>file_name</i>	::=	<i>DQSTRING</i>
<i>group_name</i>	::=	<i>IDENTIFIER</i>
<i>heap_name</i>	::=	<i>section_name</i>
<i>mem_name</i>	::=	<i>IDENTIFIER</i>
<i>proc_name</i>	::=	<i>IDENTIFIER</i>
<i>section_name</i>	::=	<i>DQSTRING</i>
<i>space_name</i>	::=	<i>IDENTIFIER</i>
<i>stack_name</i>	::=	<i>section_name</i>
<i>symbol_name</i>	::=	<i>DQSTRING</i>


```

tag_attr      ::= (tag<,tag>=>0)
tag           ::= tag = DQSTRING

```

A tag is an arbitrary text that can be added to a statement.

11.2.4. Expressions

The expressions and operators in this section work the same as in ISO C.

```

number        ::= OCT_NUM
               | DEC_NUM
               | HEX_NUM

expr          ::= number
               | symbol_name
               | unary_op expr
               | expr binary_op expr
               | expr ? expr : expr
               | ( expr )
               | function_call

unary_op      ::= !      // logical NOT
               | ~      // bitwise complement
               | -      // negative value

binary_op     ::= ^      // exclusive OR
               | *      // multiplication
               | /      // division
               | %      // modulus
               | +      // addition
               | -      // subtraction
               | >>    // right shift
               | <<    // left shift
               | ==     // equal to
               | !=     // not equal to
               | >      // greater than
               | <      // less than
               | >=     // greater than or equal to
               | <=     // less than or equal to
               | &      // bitwise AND
               | |      // bitwise OR
               | &&     // logical AND
               | ||     // logical OR

```

11.2.5. Built-in Functions

```

function_call ::= absolute ( expr )
               | addressof ( addr_id )
               | exists ( section_name )
               | max ( expr , expr )

```

TASKING VX-toolset for PCP User Guide

```
        | min ( expr , expr )  
        | sizeof ( size_id )  
  
addr_id      ::= sect : section_name  
              | group : group_name  
  
size_id      ::= sect : section_name  
              | group : group_name  
              | mem  : mem_name
```

- Every space, bus, memory, section or group you refer to, must be defined in the LSL file.
- The `addressof()` and `sizeof()` functions with the **group** or **sect** argument can only be used in the right hand side of an assignment. The `sizeof()` function with the **mem** argument can be used anywhere in section layouts.

You can use the following built-in functions in expressions. All functions return a numerical value. This value is a 64-bit signed integer.

absolute()

```
int absolute( expr )
```

Converts the value of `expr` to a positive integer.

```
absolute( "labelA"- "labelB" )
```

addressof()

```
int addressof( addr_id )
```

Returns the address of `addr_id`, which is a named section or group. To get the offset of the section with the name `asect`:

```
addressof( sect: "asect" )
```

This function only works in assignments.

exists()

```
int exists( section_name )
```

The function returns 1 if the section `section_name` exists in one or more object file, 0 otherwise. If the section is not present in input object files, but generated from LSL, the result of this function is undefined.

To check whether the section `mysection` exists in one of the object files that is specified to the linker:

```
exists( "mysection" )
```

max()

```
int max( expr, expr )
```

Returns the value of the expression that has the largest value. To get the highest value of two symbols:

```
max( "sym1" , "sym2" )
```

min()

```
int min( expr, expr )
```

Returns the value of the expression that has the smallest value. To get the lowest value of two symbols:

```
min( "sym1" , "sym2" )
```

sizeof()

```
int sizeof( size_id )
```

Returns the size of the object (group, section or memory) the identifier refers to. To get the size of the section "asection":

```
sizeof( sect: "asection" )
```

The **group** and **sect** arguments only works in assignments. The **mem** argument can be used anywhere in section layouts.

11.2.6. LSL Definitions in the Linker Script File

```
description ::= <definition>*>=1
```

```
definition ::= architecture_definition
                | derivative_definition
                | board_spec
                | section_definition
                | section_setup
```

- At least one *architecture_definition* must be present in the LSL file.

11.2.7. Memory and Bus Definitions

```
mem_def ::= memory mem_name <tag_attr>0|1 { <mem_descr ;>*>=0 }
```

- A *mem_def* defines a memory with the *mem_name* as a unique name.

```
mem_descr ::= type = <reserved>0|1 mem_type
                | mau = expr
                | size = expr
                | speed = number
                | fill <= fill_values>0|1
```

TASKING VX-toolset for PCP User Guide

```
| write_unit = expr  
| mapping
```

- A *mem_def* contains exactly one **type** statement.
- A *mem_def* contains exactly one **mau** statement (non-zero size).
- A *mem_def* contains exactly one **size** statement.
- A *mem_def* contains zero or one **speed** statement (if absent, the default speed value is 1).
- A *mem_def* contains zero or one **fill** statement.
- A *mem_def* contains zero or one **write_unit** statement.
- A *mem_def* contains at least one *mapping*

```
mem_type ::= rom // attrs = rx  
| ram // attrs = rw  
| nvram // attrs = rwx  
| blockram
```

```
fill_values ::= expr  
| [ expr <, expr>>=0 ]
```

```
bus_def ::= bus bus_name { <bus_descr ;>>=0 }
```

- A *bus_def* statement defines a bus with the given *bus_name* as a unique name within a core architecture.

```
bus_descr ::= mau = expr  
| width = expr // bus width, nr  
| // of data bits  
| mapping // legal destination  
| // 'bus' only
```

- The **mau** and **width** statements appear exactly once in a *bus_descr*. The default value for **width** is the **mau** size.
- The bus width must be an integer times the bus MAU size.
- The MAU size must be non-zero.
- A bus can only have a *mapping* on a destination bus (through **dest = bus:**).

```
mapping ::= map ( map_descr <, map_descr>>=0 )
```

```
map_descr ::= dest = destination  
| dest_dbits = range  
| dest_offset = expr  
| size = expr  
| src_dbits = range
```

```

| src_offset = expr
| tag

```

- A *mapping* requires at least the **size** and **dest** statements.
- Each *map_descr* can occur only once.
- You can define multiple mappings from a single source.
- Overlap between source ranges or destination ranges is not allowed.
- If the **src_dbits** or **dest_dbits** statement is not present, its value defaults to the **width** value if the source/destination is a bus, and to the **mau** size otherwise.

```

destination ::= space : space_name
                | bus : <proc_name |
                    core_name :>0|1 bus_name

```

- A *space_name* refers to a defined address space.
- A *proc_name* refers to a defined processor.
- A *core_name* refers to a defined core.
- A *bus_name* refers to a defined bus.
- The following mappings are allowed (source to destination)
 - space => space
 - space => bus
 - bus => bus
 - memory => bus

```

range ::= expr .. expr

```

- With address ranges, the end address is not part of the range.

11.2.8. Architecture Definition

```

architecture_definition ::= architecture arch_name
                             <( parameter_list )>0|1
                             <extends arch_name
                             <( argument_list )>0|1 >0|1
                             { <arch_spec>>=0 }

```

- An *architecture_definition* defines a core architecture with the given *arch_name* as a unique name.

TASKING VX-toolset for PCP User Guide

- At least one *space_def* and at least one *bus_def* have to be present in an *architecture_definition*.
- An *architecture_definition* that uses the **extends** construct defines an architecture that inherits all elements of the architecture defined by the second *arch_name*. The parent architecture must be defined in the LSL file as well.

```
parameter_list ::= parameter <, parameter>*>=0
parameter      ::= IDENTIFIER <= expr>0|1
argument_list  ::= expr <, expr>*>=0
arch_spec      ::= bus_def
                | space_def
                | endianness_def
space_def      ::= space space_name <tag_attr>0|1 { <space_descr;*>=0 }
```

- A *space_def* defines an address space with the given *space_name* as a unique name within an architecture.

```
space_descr    ::= space_property ;
                | section_definition //no space ref
                | vector_table_statement
                | reserved_range
space_property ::= id = number // as used in object
                | mau = expr
                | align = expr
                | page_size = expr <[ range ] <| [ range ]>*>=0>0|1
                | page
                | direction = direction
                | stack_def
                | heap_def
                | copy_table_def
                | start_address
                | mapping
```

- A *space_def* contains exactly one **id** and one **mau** statement.
- A *space_def* contains at most one **align** statement.
- A *space_def* contains at most one **page_size** statement.
- A *space_def* contains at most 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
                  | id = expr
                  | tag
```

- The **min_size** statement must be present.
- You can specify at most one **align** statement and one **grows** statement.
- Each stack definition has its own unique **id**, the number specified corresponds to the index in the **.CALLS** directive as generated by the compiler.

```
direction ::= low_to_high
           | high_to_low
```

- If you do not specify the **grows** statement, the stack and heap grow **low-to-high**.

```
copy_table_def ::= copytable <( copy_table_descr
                                <, copy_table_descr > >=0 ) >0|1
```

- A *space_def* contains at most one **copytable** statement.
- Exactly one copy table must be defined in one of the spaces.

```
copy_table_descr ::= align = expr
                  | copy_unit = expr
                  | dest <space_name>0|1 = space_name
                  | page
                  | tag
```

- The **copy_unit** is defined by the size in MAUs in which the startup code moves data.
- The **dest** statement is only required when the startup code initializes memory used by another processor that has no access to ROM.
- A *space_name* refers to a defined address space.

```
start_addr ::= start_address ( start_addr_descr
                              <, start_addr_descr > >=0 )
```

```
start_addr_descr ::= run_addr = expr
                  | symbol = symbol_name
```

- A *symbol_name* refers to the section that contains the startup code.

```

vector_table_statement
    ::= vector_table section_name
       ( vecttab_spec <, vecttab_spec>*>=0 )
       { <vector_def>*>=0 }

vecttab_spec
    ::= vector_size = expr
       | size = expr
       | id_symbol_prefix = symbol_name
       | run_addr = addr_absolute
       | template = section_name
       | template_symbol = symbol_name
       | vector_prefix = section_name
       | fill = vector_value
       | no_inline
       | copy
       | tag

vector_def
    ::= vector ( vector_spec <, vector_spec>*>=0 );

vector_spec
    ::= id = vector_id_spec
       | fill = vector_value
       | optional
       | tag

vector_id_spec
    ::= number
       | [ range ] <, [ range ]>*>=0

vector_value
    ::= symbol_name
       | [ number <, number>*>=0 ]
       | loop <[ expr ]>*>=0|1

reserved_range
    ::= reserved <tag_attr>*>=0|1 expr .. expr ;

```

- The end address is not part of the range.

```

endianness_def
    ::= endianness { <endianness_type;>*>=1 }

endianness_type
    ::= big
       | little

```

11.2.9. Derivative Definition

```

derivative_definition
    ::= derivative derivative_name
       <( parameter_list )>*>=0|1
       <extends derivative_name
          <( argument_list )>*>=0|1 >*>=0|1
          { <derivative_spec>*>=0 }

```

- A *derivative_definition* defines a derivative with the given *derivative_name* as a unique name.


```

derivative_spec ::= core_def
                  | bus_def
                  | mem_def
                  | section_definition // no processor name
                  | section_setup

```

```

core_def ::= core core_name { <core_descr ;>>=0 }

```

- A *core_def* defines a core with the given *core_name* as a unique name.
- At least one *core_def* must be present in a *derivative_definition*.

```

core_descr ::= architecture = arch_name
                <( argument_list )>0|1
                | endianness = ( endianness_type
                                   <, endianness_type>>=0 )

```

- An *arch_name* refers to a defined core architecture.
- Exactly one **architecture** statement must be present in a *core_def*.

11.2.10. Processor Definition and Board Specification

```

board_spec ::= proc_def
                | bus_def
                | mem_def

```

```

proc_def ::= processor proc_name
              { proc_descr ; }

```

```

proc_descr ::= derivative = derivative_name
                <( argument_list )>0|1

```

- A *proc_def* defines a processor with the *proc_name* as a unique name.
- If you do not explicitly define a processor for a derivative in an LSL file, the linker defines a processor with the same name as that derivative.
- A *derivative_name* refers to a defined derivative.
- A *proc_def* contains exactly one **derivative** statement.

11.2.11. Section Layout Definition and Section Setup

```

section_definition ::= section_layout <space_ref>0|1
                        <( space_layout_properties )>0|1
                        { <section_statement>>=0 }

```

- A section definition inside a space definition does not have a *space_ref*.
- All global section definitions have a *space_ref*.

TASKING VX-toolset for PCP User Guide

```
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.

```
space_layout_properties ::= space_layout_property <, space_layout_property >=0
```

```
space_layout_property ::= locate_direction  
                       | tag
```

```
locate_direction ::= direction = direction
```

```
direction ::= low_to_high  
           | high_to_low
```

- A section layout contains at most one **direction** statement.
- If you do not specify the **direction** statement, the locate direction of the section layout is **low-to-high**.

```
section_statement ::= simple_section_statement ;  
                  | aggregate_section_statement
```

```
simple_section_statement ::= assignment  
                        | select_section_statement  
                        | special_section_statement
```

```
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_selection
    ::= ( section_selection
         <, section_selection>*> )

```

```

section_selection
    ::= attributes = < <+|-> attribute>*>
       | tag

```

- **+attribute** means: select all sections that have this attribute.
- **-attribute** means: select all sections that do not have this attribute.

```

special_section_statement
    ::= heap heap_name <stack_heap_mods>*>
       | stack stack_name <stack_heap_mods>*>
       | copytable
       | reserved section_name <reserved_specs>*>

```

- Special sections cannot be selected in load-time groups.

```

stack_heap_mods    ::= ( stack_heap_mod <, stack_heap_mod>*> )

```

```

stack_heap_mod     ::= size = expr
                   | tag

```

```

reserved_specs     ::= ( reserved_spec <, reserved_spec>*> )

```

```

reserved_spec      ::= attributes
                   | fill_spec
                   | size = expr
                   | alloc_allowed = absolute | ranged

```

- If a **reserved** section has attributes **r**, **rw**, **x**, **rx** or **rwX**, and no fill pattern is defined, the section is filled with zeros. If no attributes are set, the section is created as a scratch section (attributes **ws**, no image).

```

fill_spec          ::= fill = fill_values

```

```

fill_values        ::= expr
                   | [ expr <, expr>*> ]

```

```

aggregate_section_statement
    ::= { <section_statement>*> }
       | group_descr
       | if_statement
       | section_creation_statement

```

```

group_descr        ::= group <group_name>*> <( group_specs )>*>
                   section_statement

```

- For every group with a name, the linker defines a label.
- No two groups for address spaces of a core can have the same *group_name*.

TASKING VX-toolset for PCP User Guide

```
group_specs      ::= group_spec <, group_spec >*>=0
group_spec       ::= group_alignment
                  | attributes
                  | copy
                  | nocopy
                  | group_load_address
                  | fill <= fill_values>^0|1
                  | group_page
                  | group_run_address
                  | group_type
                  | allow_cross_references
                  | priority = number
                  | tag
```

- The **allow-cross-references** property is only allowed for *overlay* groups.
- Sub groups inherit all properties from a parent group.

```
group_alignment  ::= align = expr
attributes       ::= attributes = <attribute>*>=1
attribute        ::= r    // readable sections
                  | w    // writable sections
                  | x    // executable code sections
                  | i    // initialized sections
                  | s    // scratch sections
                  | b    // blanked (cleared) sections
group_load_address ::= load_addr <= load_or_run_addr>^0|1
group_page       ::= page <= expr>^0|1
                  | page_size = expr <[ range ] <| [ range ]>*>=0>^0|1
group_run_address ::= run_addr <= load_or_run_addr>^0|1
group_type       ::= clustered
                  | contiguous
                  | ordered
                  | overlay
```

- For *non-contiguous* groups, you can only specify *group_alignment* and *attributes*.
- The **overlay** keyword also sets the **contiguous** property.
- The **clustered** property cannot be set together with **contiguous** or **ordered** on a single group.

```
load_or_run_addr ::= addr_absolute
                  | addr_range <| addr_range>*>=0
```

```
addr_absolute ::= expr
               | memory_reference [ expr ]
```

- An absolute address can only be set on *ordered* groups.

```
addr_range ::= [ expr .. expr ]
            | memory_reference
            | memory_reference [ expr .. expr ]
```

- The parent of a group with an *addr_range* or **page** restriction cannot be **ordered**, **contiguous** or **clustered**.
- The end address is not part of the range.

```
memory_reference ::= mem : <proc_name :>0|1 mem_name
```

- A *proc_name* refers to a defined processor.
- 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
              | tag
```

```
section_statement2
 ::= select_section_statement ;
    | group_descr2
    | { <section_statement2>>=0 }
```

```
group_descr2 ::= group <group_name>0|1
              ( group_specs2 )
              section_statement2
```

```
group_specs2 ::= group_spec2 <, group_spec2 >>=0
```

```
group_spec2 ::= group_alignment
              | attributes
              | load_addr
              | tag
```

```
section_setup      ::= section_setup space_ref <tag_attr>0|1
                    { <section_setup_item>>=0 }

section_setup_item
                    ::= vector_table_statement
                       | reserved_range
                       | stack_def ;
                       | heap_def ;
```

11.3. Expression Evaluation

Only *constant* expressions are allowed, including sizes, but not addresses, of sections in object files.

All expressions are evaluated with 64-bit precision integer arithmetic. The result of an expression can be absolute or relocatable. A symbol you assign is created as an absolute symbol.

11.4. Semantics of the Architecture Definition

Keywords in the architecture definition

```
architecture
  extends
endianness      big little
bus
  mau
  width
  map
space
  id
  mau
  align
  page_size
  page
  direction      low_to_high high_to_low
  stack
    min_size
    grows         low_to_high high_to_low
    align
    fixed
    id
  heap
    min_size
    grows         low_to_high high_to_low
    align
    fixed
    id
copytable
  align
  copy_unit
  dest
```

```

    page
vector_table
    vector_size
    size
    id_symbol_prefix
    run_addr
    template
    template_symbol
    vector_prefix
    fill
    no_inline
    copy
    vector
        id
            fill        loop
            optional
reserved
start_address
    run_addr
    symbol
map

map
    dest            bus    space
    dest_dbits
    dest_offset
    size
    src_dbits
    src_offset

```

11.4.1. Defining an Architecture

With the keyword **architecture** you define an architecture and assign a unique name to it. The name is used to refer to it at other places in the LSL file:

```

architecture name
{
    definitions
}

```

If you are defining multiple core architectures that show great resemblance, you can define the common features in a parent core architecture and extend this with a child core architecture that contains specific features. The child inherits all features of the parent. With the keyword **extends** you create a child core architecture:

```

architecture name_child_arch extends name_parent_arch
{
    definitions
}

```

A core architecture can have any number of parameters. These are identifiers which get values assigned on instantiation or extension of the architecture. You can use them in any expression within the core architecture. Parameters can have default values, which are used when the core architecture is instantiated with less arguments than there are parameters defined for it. When you extend a core architecture you can pass arguments to the parent architecture. Arguments are expressions that set the value of the parameters of the sub-architecture.

```
architecture name_child_arch (parm1,parm2=1)
    extends name_parent_arch (arguments)
{
    definitions
}
```

11.4.2. Defining Internal Buses

With the **bus** keyword you define a bus (the combination of data and corresponding address bus). The bus name is used to identify a bus and does not conflict with other identifiers. Bus descriptions in an architecture definition or derivative definition define *internal* buses. Some internal buses are used to communicate with the components outside the core or processor. Such buses on a processor have physical pins reserved for the number of bits specified with the **width** statements.

- The **mau** field specifies the MAU size (Minimum Addressable Unit) of the data bus. This field is required.
- The **width** field specifies the width (number of address lines) of the data bus. The default value is the MAU size.
- The **map** keyword specifies how this bus maps onto another bus (if so). Mappings are described in [Section 11.4.4, *Mappings*](#).

```
bus bus_name
{
    mau = 8;
    width = 8;
    map ( map_description );
}
```

11.4.3. Defining Address Spaces

With the **space** keyword you define a logical address space. The space name is used to identify the address space and does not conflict with other identifiers.

- The **id** field defines how the addressing space is identified in object files. In general, each address space has a unique ID. The linker locates sections with a certain ID in the address space with the same ID. This field is required.
- The **mau** field specifies the MAU size (Minimum Addressable Unit) of the space. This field is required.
- The **align** value must be a power of two. The linker uses this value to compute the start addresses when sections are concatenated. An align value of *n* means that objects in the address space have to be aligned on *n* MAUs.

- The `page_size` field sets the page alignment and page size in MAUs for the address space. It must be a power of 2. The default value is 1. If one or more page ranges are supplied the supplied value only sets the page alignment. The ranges specify the available space in each page, as offsets to the page start, which is aligned at the page alignment.

See also the `page` keyword in subsection [Locating a group](#) in [Section 11.8.2, *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 11.4.4, *Mappings*](#).

Stacks and heaps

- The `stack` keyword defines a stack in the address space and assigns a name to it. The architecture definition must contain at least one stack definition. Each stack of a core architecture must have a unique name. See also the `stack` keyword in [Section 11.8.3, *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.

The `id` keyword matches stack information generated by the compiler with a stack name specified in LSL. This value assigned to this keyword is strongly related to the compiler's output, so users are not supposed to change this configuration.

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 11.8.3, *Creating or Modifying Special Sections*](#).

Stacks and heaps are only generated by the linker if the corresponding linker labels are referenced in the object files.

See [Section 11.8, *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.

Vector table

- The `vector_table` keyword defines a vector table with n vectors of size m (This is an internal LSL object similar to an LSL group.) The `run_addr` argument specifies the location of the first vector (`id=0`). This can be a simple address or an offset in memory (see the description of the run-time address in subsection [Locating a group](#) in [Section 11.8.2, Creating and Locating Groups of Sections](#)). A vector table defines symbols `_lc_ub_foo` and `_lc_ue_foo` pointing to start and end of the table.

```
vector_table "vtable" (vector_size=m, size=n, run_addr=x, ...)
```

See the following example of a vector table definition:

```
vector_table "vtable" (vector_size = 4, size = 256, run_addr=0,
    template=".text.vector_template",
    template_symbol="_lc_vector_target",
    vector_prefix=".text.vector.",
    id_symbol_prefix="foo",
    no_inline,
    /* default: empty, or */
    fill="foo", /* or */
    fill=[1,2,3,4], /* or */
    fill=loop)
{
    vector (id=23, fill="main", optional);
    vector (id=12, fill=[0xab, 0x21, 0x32, 0x43]);
    vector (id=[1..11], fill=[0]);
    vector (id=[18..23], fill=loop);
}
```

The `template` argument defines the name of the section that holds the code to jump to a handler function from the vector table. This template section does not get located and is removed when the locate phase is completed. This argument is required.

The `template_symbol` argument is the symbol reference in the template section that must be replaced by the address of the handler function. This symbol name should start with the linker prefix for the symbol to be ignored in the link phase. This argument is required.

The `vector_prefix` argument defines the names of vector sections: the section for a vector with id `vector_id` is `$(vector_prefix)$$(vector_id)`. Vectors defined in C or assembly source files that should be included in the vector table must have the correct symbol name. The compiler uses the prefix that is defined in the default LSL file(s); if this attribute is changed, the vectors declared in C source files are not included in the vector table. When a vector supplied in an object file has exactly one relocation, the linker will assume it is a branch to a handler function, and can be removed when the handler is inlined in the vector table. Otherwise, no inlining is done. This argument is required.

With the optional `no_inline` argument the vectors handlers are not inlined in the vector table.

With the optional `copy` argument a ROM copy of the vector table is made and the vector table is copied to RAM at startup.

With the optional `id_symbol_prefix` argument you can set an internal string representing a symbol name prefix that may be found on symbols in vector handler code. When the linker detects such a symbol in a handler, the symbol is assigned the vector number. If the symbol was already assigned a vector number, a warning is issued.

The `fill` argument sets the default contents of vectors. If nothing is specified for a vector, this setting is used. See below. When no default is provided, empty vectors may be used to locate large vector handlers and other sections. Only one `fill` argument is allowed.

The `vector` field defines the content of vector with the number specified by `id`. If a range is specified for `id` (`[p . . q, s . . t]`) all vectors in the ranges (inclusive) are defined the same way.

With `fill=symbol_name`, the vector must jump to this symbol. If the section in which the symbol is defined fits in the vector table (size may be $>m$), locate the section at the location of the vector. Otherwise, insert code to jump to the symbol's value. A template interrupt handler section name + symbol name for the target code must be supplied in the LSL file.

`fill=[value(s)]`, fills the vector with the specified MAU values.

With `fill=loop` the vector jumps to itself. With the optional `[offset]` you can specify an offset from the vector table entry.

When the keyword `optional` is set on a vector specification with a symbol value and the symbol is not found, no error is reported. A default fill value is used if the symbol was not found. With other values the attribute has no effect.

Reserved address ranges

- The `reserved` keyword specifies to reserve a part of an address space even if not all of the range is covered by memory. See also the `reserved` keyword in [Section 11.8.3, Creating or Modifying Special Sections](#).

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.

TASKING VX-toolset for PCP User Guide

The `run_addr` argument specifies the start address (reset vector). If the core starts executing using an entry from a vector table, and directly jumps to the start label, you should omit this argument.

The `symbol` argument specifies the name of the label in the application code that should be located at the specified start address. The `symbol` argument is required. The linker will resolve the start symbol and use its value after locating for the start address field in IEEE-695 files and Intel Hex files. If you also specified the `run_addr` argument, the start symbol (label) must point to a section. The linker locates this section such that the start symbol ends up on the start address.

```
space space_name
{
    id = 1;
    mau = 8;
    align = 8;
    page_size = 1;
    stack name (min_size = 1k, grows = low_to_high);
    reserved start_address .. end_address;
    start_address ( run_addr = 0x0000,
                   symbol = "start_label" )
    map ( map_description );
}
```

11.4.4. Mappings

You can use a mapping when you define a space, bus or memory. With the `map` field you specify how addresses from the source (space, bus or memory) are translated to addresses of a destination (space, bus). The following mappings are possible:

- space => space
- space => bus
- bus => bus
- memory => bus

With a mapping you specify a range of source addresses you want to map (specified by a source offset and a size), the destination to which you want to map them (a bus or another address space), and the offset address in the destination.

- The `dest` argument specifies the destination. This can be a `bus` or another address `space` (only for a space to space mapping). This argument is required.
- The `src_offset` argument specifies the offset of the source addresses. In combination with size, this specifies the range of address that are mapped. By default the source offset is 0x0000.
- The `size` argument specifies the number of addresses that are mapped. This argument is required.
- The `dest_offset` argument specifies the position in the destination to which the specified range of addresses is mapped. By default the destination offset is 0x0000.

If you are mapping a bus to another bus, the number of data lines of each bus may differ. In this case you have to specify a range of source data lines you want to map (`src_dbits = begin..end`) and the range of destination data lines you want to map them to (`dest_dbits = first..last`).

- The `src_dbits` argument specifies a range of data lines of the source bus. By default all data lines are mapped.
- The `dest_dbits` argument specifies a range of data lines of the destination bus. By default, all data lines from the source bus are mapped on the data lines of the destination bus (starting with line 0).

From space to space

If you map an address space to another address space (nesting), you can do this by mapping the subspace to the containing larger space. In this example a small space of 64 kB is mapped on a large space of 16 MB.

```
space small
{
    id = 2;
    mau = 4;
    map (src_offset = 0, dest_offset = 0,
        dest = space : large, size = 64k);
}
```

From space to bus

All spaces that are not mapped to another space must map to a bus in the architecture:

```
space large
{
    id = 1;
    mau = 4;
    map (src_offset = 0, dest_offset = 0,
        dest = bus:bus_name, size = 16M );
}
```

From bus to bus

The next example maps an external bus called `e_bus` to an internal bus called `i_bus`. This internal bus resides on a core called `mycore`. The source bus has 16 data lines whereas the destination bus has only 8 data lines. Therefore, the keywords `src_dbits` and `dest_dbits` specify which source data lines are mapped on which destination data lines.

```
architecture mycore
{
    bus i_bus
    {
        mau = 4;
    }

    space i_space
```

```
    {
        map (dest=bus:i_bus, size=256);
    }
}

bus e_bus
{
    mau = 16;
    width = 16;
    map (dest = bus:mycore:i_bus, src_dbits = 0..7, dest_dbits = 0..7 )
}
```

It is not possible to map an internal bus to an external bus.

11.5. Semantics of the Derivative Definition

Keywords in the derivative definition

```
derivative
    extends
core
    architecture
bus
    mau
    width
    map
memory
    type          reserved rom ram nvram blockram
    mau
    size
    speed
    fill
    write_unit
    map
section_layout
section_setup

    map
        dest          bus space
        dest_dbits
        dest_offset
        size
        src_dbits
        src_offset
```

11.5.1. Defining a Derivative

With the keyword **derivative** you define a derivative and assign a unique name to it. The name is used to refer to it at other places in the LSL file:

```

derivative name
{
    definitions
}

```

If you are defining multiple derivatives that show great resemblance, you can define the common features in a parent derivative and extend this with a child derivative that contains specific features. The child inherits all features of the parent (cores and memories). With the keyword **extends** you create a child derivative:

```

derivative name_child_deriv extends name_parent_deriv
{
    definitions
}

```

As with a core architecture, a derivative can have any number of parameters. These are identifiers which get values assigned on instantiation or extension of the derivative. You can use them in any expression within the derivative definition.

```

derivative name_child_deriv (parm1,parm2=1)
    extends name_parent_deriv (arguments)
{
    definitions
}

```

11.5.2. Instantiating Core Architectures

With the keyword **core** you instantiate a core architecture in a derivative.

- With the keyword **architecture** you tell the linker that the given core has a certain architecture. The architecture name refers to an existing architecture definition in the same LSL file.

For example, if you have two cores (called `mycore_1` and `mycore_2`) that have the same architecture (called `mycorearch`), you must instantiate both cores as follows:

```

core mycore_1
{
    architecture = mycorearch;
}

core mycore_2
{
    architecture = mycorearch;
}

```

If the architecture definition has parameters you must specify the arguments that correspond with the parameters. For example `mycorearch1` expects two parameters which are used in the architecture definition:

```

core mycore
{

```

```
    architecture = mycorearch1 (1,2);  
}
```

11.5.3. 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 11.6.3, Defining External Memory and Buses](#)).

- The **type** field specifies a memory type:
 - **rom**: read-only memory - it can only be written at load-time
 - **ram**: random access volatile writable memory - writing at run-time is possible while writing at load-time has no use since the data is not retained after a power-down
 - **nvr****ram**: non volatile ram - writing is possible both at load-time and run-time
 - **blockram**: writing is possible both at load-time and run-time. Changes are applied in RAM, so after a full device reset the data in a blockram reverts to the original state.

The optional **reserved** qualifier before the memory type, tells the linker not to locate any section in the memory by default. You can locate sections in such memories using an absolute address or range restriction (see subsection [Locating a group](#) in [Section 11.8.2, 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 (1..4): 1 is the slowest, 4 the fastest. The linker uses the relative speed of the memories in such a way, that faster memory is used before slower memory. The default speed is 1.
- The **map** field specifies how this memory maps onto an (internal) bus. Mappings are described in [Section 11.4.4, Mappings](#).
- The optional **write_unit** field specifies the minimum write unit (MWU). This is the minimum number of MAUs required in a write action. This is useful to initialize memories that can only be written in units of two or more MAUs. If **write_unit** is not defined the minimum write unit is 0.
- The optional **fill** field contains a bit pattern that the linker writes to all memory addresses that remain unoccupied during the locate process. The result of the expression, or list of expressions, is used as values to write to memory, each in MAU.

```
memory mem_name  
{  
    type = rom;  
    mau = 8;  
    write_unit = 4;  
    fill = 0xaa;
```



```

    size = 64k;
    speed = 2;
    map ( map_description );
}

```

With the `bus` keyword you define a bus in a derivative definition. Buses are described in [Section 11.4.2, Defining Internal Buses](#).

11.6. Semantics of the Board Specification

Keywords in the board specification

```

processor
  derivative
bus
  mau
  width
  map
memory
  type          reserved rom ram nvram blockram
  mau
  size
  speed
  fill
  write_unit
  map

  map
    dest          bus space
    dest_dbits
    dest_offset
    size
    src_dbits
    src_offset

```

11.6.1. Defining a Processor

If you have a target board with multiple processors that have the same derivative, you need to instantiate each individual processor in a processor definition. This information tells the linker which processor has which derivative and enables the linker to distinguish between the present processors.

If you use processors that all have a unique derivative, you may omit the processor definitions. In this case the linker assumes that for each derivative definition in the LSL file there is one processor. The linker uses the derivative name also for the processor.

With the keyword `processor` you define a processor. You can freely choose the processor name. The name is used to refer to it at other places in the LSL file:

```
processor proc_name
{
    processor definition
}
```

11.6.2. Instantiating Derivatives

With the keyword **derivative** you tell the linker that the given processor has a certain derivative. The derivative name refers to an existing derivative definition in the same LSL file.

For example, if you have two processors on your target board (called `myproc_1` and `myproc_2`) that have the same derivative (called `myderiv`), you must instantiate both processors as follows:

```
processor myproc_1
{
    derivative = myderiv;
}

processor myproc_2
{
    derivative = myderiv;
}
```

If the derivative definition has parameters you must specify the arguments that correspond with the parameters. For example `myderiv1` expects two parameters which are used in the derivative definition:

```
processor myproc
{
    derivative = myderiv1 (2,4);
}
```

11.6.3. Defining External Memory and Buses

It is common to define external memory (off-chip) and external buses at the global scope (outside any enclosing definition). Internal memory (on-chip memory) is usually defined in the scope of a derivative definition.

With the keyword **memory** you define physical memory that is present on the target board. The memory name is used to identify the memory and does not conflict with other identifiers. If you define memory parts in the LSL file, only the memory defined in these parts is used for placing sections.

If no external memory is defined in the LSL file and if the linker option to allocate memory on demand is set then the linker will assume that all virtual addresses are mapped on physical memory. You can override this behavior by specifying one or more memory definitions.

```
memory mem_name
{
    type = rom;
    mau = 8;
    write_unit = 4;
    fill = 0xaa;
```

```

    size = 64k;
    speed = 2;
    map ( map_description );
}

```

For a description of the keywords, see [Section 11.5.3, *Defining Internal Memory and Buses*](#).

With the keyword `bus` you define a bus (the combination of data and corresponding address bus). The bus name is used to identify a bus and does not conflict with other identifiers. Bus descriptions at the global scope (outside any definition) define external buses. These are buses that are present on the target board.

```

bus bus_name
{
    mau = 8;
    width = 8;
    map ( map_description );
}

```

For a description of the keywords, see [Section 11.4.2, *Defining Internal Buses*](#).

You can connect off-chip memory to any derivative: you need to map the off-chip memory to a bus and map that bus on the internal bus of the derivative you want to connect it to.

11.7. Semantics of the Section Setup Definition

Keywords in the section setup definition

```

section_setup
  stack
    min_size
    grows          low_to_high  high_to_low
    align
    fixed
    id
  heap
    min_size
    grows          low_to_high  high_to_low
    align
    fixed
    id
  vector_table
    vector_size
    size
    id_symbol_prefix
    run_addr
    template
    template_symbol
    vector_prefix
    fill
    no_inline

```

```
copy
vector
  id
  fill      loop
  optional
reserved
```

11.7.1. Setting up a Section

With the keyword **section_setup** you can define stacks, heaps, vector tables, and/or reserved address ranges outside their address space definition.

```
section_setup ::my_space
{
  vector table statements
  reserved address range
  stack definition
  heap definition
}
```

See the subsections [Stacks and heaps](#), [Vector table](#) and [Reserved address ranges](#) in [Section 11.4.3, Defining Address Spaces](#) for details on the keywords **stack**, **heap**, **vector_table** and **reserved**.

11.8. 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
  copy
  nocopy
  fill
  ordered
  contiguous
  clustered
  overlay
  allow_cross_references
  load_addr
  mem
  run_addr
  mem
  page
  page_size
  priority
select
stack
  size
```

```

heap
  size
reserved
  size
  attributes    r w x
  fill
  alloc_allowed absolute ranged
copytable
section
  size
  blocksize
  attributes    r w x
  fill
  overflow

if
else

```

11.8.1. 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.

11.8.2. 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 11.8.3, Creating or Modifying Special Sections](#).

With the *group_specifications* you actually locate the sections in the group. This is described in subsection [Locating a group](#).

Selecting sections for a group

With the keyword **select** you can select one or more sections for the group. You can select a section by name or by attributes. If you select a section by name, you can use a wildcard pattern:

- * matches with all section names
- ? matches with a single character in the section name
- \ takes the next character literally
- [abc] matches with a single 'a', 'b' or 'c' character
- [a-z] matches with any single character in the range 'a' to 'z'

```
group ( ... )
{
    select "mysection";
    select "*";
}
```

The first **select** statement selects the section with the name "mysection". The second **select** statement selects all sections that were not selected yet.

A section is selected by the first select statement that matches, in the union of all section layouts for the address space. Global section layouts are processed in the order in which they appear in the LSL file. Internal core architecture section layouts always take precedence over global section layouts.

- The **attributes** field selects all sections that carry (or do not carry) the given attribute. With **+attribute** you select sections that have the specified attribute set. With **-attribute** you select sections that do not have the specified attribute set. You can specify one or more of the following attributes:

- **r** readable sections

- **w** writable sections
- **x** executable sections
- **i** initialized sections
- **b** sections that should be cleared at program startup
- **s** scratch sections (not cleared and not initialized)

To select all read-only sections:

```
group ( ... )
{
    select (attributes = +r-w);
}
```

Keep in mind that all section selections are restricted to the address space of the section layout in which this group definition occurs.

- With the **ref_tree** field you can select a group of related sections. The relation between sections is often expressed by means of references. By selecting just the 'root' of tree, the complete tree is selected. This is for example useful to locate a group of related sections in special memory (e.g. fast memory). The (referenced) sections must meet the following conditions in order to be selected:
 1. The sections are within the section layout's address space
 2. The sections match the specified attributes
 3. The sections have no absolute restriction (as is the case for all wildcard selections)

For example, to select the code sections referenced from `foo1`:

```
group refgrp (ordered, contiguous, run_addr=mem:ext_c)
{
    select ref_tree "foo1" (attributes==+x);
}
```

If section `foo1` references `foo2` and `foo2` references `foo3`, then all these sections are selected by the selection shown above.

Locating a group

```
group group_name ( group_specifications )
{
    section_statements
}
```

With the *group_specifications* you actually define how the linker must locate the group. You can roughly define three things: 1) assign properties to the group like alignment and read/write attributes, 2) define the mutual order in the address space for sections in the group and 3) restrict the possible addresses for the sections in a group.

TASKING VX-toolset for PCP User Guide

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 overrides the default attributes. By default the linker uses the attributes of the input sections. You can set the **r**, **w**, or **rw** attributes and you can switch between the **b** and **s** attributes.
- The **copy** field tells the linker to locate a read-only section in RAM and generate a ROM copy and a copy action in the copy table. This property makes the sections in the group writable which causes the linker to generate ROM copies for the sections.
- The effect of the **nocopy** field is the opposite of the **copy** field. It prevents the linker from generating ROM copies of the selected sections.

2. Define the mutual order of the sections in the group.

By default, a group is *unrestricted* which means that the linker has total freedom to place the sections of the group in the address space.

- The **ordered** keyword tells the linker to locate the sections in the same order in the address space as they appear in the group (but not necessarily adjacent).

Suppose you have an ordered group that contains the sections 'A', 'B' and 'C'. By default the linker places the sections in the address space like 'A' - 'B' - 'C', where section 'A' gets the lowest possible address. With **direction=high_to_low** in the **section_layout** space properties, the linker places the sections in the address space like 'C' - 'B' - 'A', where section 'A' gets the highest possible address.

- The **contiguous** keyword tells the linker to locate the sections in the group in a single address range. Within a contiguous group the input sections are located in arbitrary order, however the group occupies one contiguous range of memory. Due to alignment of sections there can be 'alignment gaps' between the sections.

When you define a group that is both **ordered** and **contiguous**, this is called a *sequential* group. In a sequential group the linker places sections in the same order in the address space as they appear in the group and it occupies a contiguous range of memory.

- The **clustered** keyword tells the linker to locate the sections in the group in a number of *contiguous* blocks. It tries to keep the number of these blocks to a minimum. If enough memory is available, the

group will be located as if it was specified as **contiguous**. Otherwise, it gets split into two or more blocks.

If a contiguous or clustered group contains *alignment gaps*, the linker can locate sections that are not part of the group in these gaps. To prevent this, you can use the **fill** keyword. If the group is located in RAM, the gaps are treated as reserved (scratch) space. If the group is located in ROM, the alignment gaps are filled with zeros by default. You can however change the fill pattern by specifying a bit pattern. The result of the expression, or list of expressions, is used as values to write to memory, each in MAU.

- The **overlay** keyword tells the linker to overlay the sections in the group. The linker places all sections in the address space using a contiguous range of addresses. (Thus an overlay group is automatically also a contiguous group.) To overlay the sections, all sections in the overlay group share the same run-time address.

For each input section within the overlay the linker automatically defines two symbols. The symbol `_lc_cb_section_name` is defined as the load-time start address of the section. The symbol `_lc_ce_section_name` is defined as the load-time end address of the section. C (or assembly) code may be used to copy the overlaid sections.

If sections in the overlay group contain references between groups, the linker reports an error. The keyword **allow_cross_references** tells the linker to accept cross-references. Normally, it does not make sense to have references between sections that are overlaid.

```
group ovl (overlay)
{
    group a
    {
        select "my_ovl_p1";
        select "my_ovl_p2";
    }
    group b
    {
        select "my_ovl_q1";
    }
}
```

It may be possible that one of the sections in the overlay group already has been defined in another group where it received a load-time address. In this case the linker does not overrule this load-time address and excludes the section from the overlay group.

3. Restrict the possible addresses for the sections in a group.

The load-time address specifies where the group's elements are loaded in memory at download time. The run-time address specifies where sections are located at run-time, that is when the program is executing. If you do not explicitly restrict the address in the LSL file, the linker assigns addresses to the sections based on the restrictions relative to other sections in the LSL file and section alignments. The program is responsible for copying overlay sections at appropriate moment from its load-time location to its run-time location (this is typically done by the startup code).

TASKING VX-toolset for PCP User Guide

- The `run_addr` keyword defines the run-time address. If the run-time location of a group is set explicitly, the given order between groups specify whether the run-time address propagates to the parent group or not. The location of the sections a group can be restricted either to a single absolute address, or to a number of address ranges (not including the end address). With an expression you can specify that the group should be located at the absolute address specified by the expression:

```
group (run_addr = 0xa00f0000)
```

You can use the `[offset]` variant to locate the group at the given absolute offset in memory:

```
group (run_addr = mem:A[0x1000])
```

A range can be an absolute space address range, written as `[expr .. expr]`, a complete memory device, written as `mem:mem_name`, or a memory address range, `mem:mem_name[expr .. expr]`

```
group (run_addr = mem:my_dram)
```

You can use the `|` to specify an address range of more than one physical memory device:

```
group (run_addr = mem:A | mem:B)
```

- The `load_addr` keyword changes the meaning of the section selection in the group: the linker selects the load-time ROM copy of the named section(s) instead of the regular sections. Just like `run_addr` you can specify an absolute address or an address range.

```
group (contiguous, load_addr)
{
    select "mydata"; // select ROM copy of mydata:
                    // "[mydata]"
}
```

The load-time and run-time addresses of a group cannot be set at the same time. If the load-time property is set for a group, the group (only) restricts the positioning at load-time of the group's sections. It is not possible to set the address of a group that has a not-unrestricted parent group.

The properties of the load-time and run-time start address are:

- At run-time, before using an element in an overlay group, the application copies the sections from their load location to their run-time location, but only if these two addresses are different. For non-overlay sections this happens at program start-up.
- The start addresses cannot be set to absolute values for unrestricted groups.
- For non-overlay groups that do not have an overlay parent, the load-time start address equals the run-time start address.
- For any group, if the run-time start address is not set, the linker selects an appropriate address.
- If an ordered group or sequential group has an absolute address and contains sections that have separate page restrictions (not defined in LSL), all those sections are located in a single page. In

other cases, for example when an unrestricted group has an address range assigned to it, the paged sections may be located in different pages.

For overlays, the linker reserves memory at the run-time start address as large as the largest element in the overlay group.

- The **page** keyword tells the linker to place the group in one page. Instead of specifying a run-time address, you can specify a page and optional a page number. Page numbers start from zero. If you omit the page number, the linker chooses a page.

The **page** keyword refers to pages in the address space as defined in the architecture definition.

- With the **page_size** keyword you can override the page alignment and size set on the address space. When you set the page size to zero, the linker removes simple (auto generated) page restrictions from the selected sections. See also the **page_size** keyword in [Section 11.4.3, Defining Address Spaces](#).
- With the **priority** keyword you can change the order in which sections are located. This is useful when some sections are considered important for good performance of the application and a small amount of fast memory is available. The value is a number for which the default is 1, so higher priorities start at 2. Sections with a higher priority are located before sections with a lower priority, unless their relative locate priority is already determined by other restrictions like **run_addr** and **page**.

```
group (priority=2)
{
    select "importantcode1";
    select "importantcode2";
}
```

11.8.3. Creating or Modifying Special Sections

Instead of selecting sections, you can also create a reserved section or an output section or modify special sections like a stack or a heap. Because you cannot define these sections in the input files, you must use the linker to create them.

Stack

- The keyword **stack** tells the linker to reserve memory for the stack. The name for the stack section refers to the stack as defined in the architecture definition. If no name was specified in the architecture definition, the default name is **stack**.

With the keyword **size** you can specify the size for the stack. If the size is not specified, the linker uses the size given by the **min_size** argument as defined for the stack in the architecture definition. Normally the linker automatically tries to maximize the size, unless you specified the keyword **fixed**.

```
group ( ... )
{
    stack "mystack" ( size = 2k );
}
```

The linker creates two labels to mark the begin and end of the stack, `_lc_ub_stack_name` for the begin of the stack and `_lc_ue_stack_name` for the end of the stack. The linker allocates space for the stack when there is a reference to either of the labels.

See also the `stack` keyword in [Section 11.4.3, Defining Address Spaces](#).

Heap

- The keyword `heap` tells the linker to reserve a dynamic memory range for the `malloc()` function. Each heap section has a name. With the keyword `size` you can change the size for the heap. If the `size` is not specified, the linker uses the size given by the `min_size` argument as defined for the heap in the architecture definition. Normally the linker automatically tries to maximize the size, unless you specified the keyword `fixed`.

```
group ( ... )
{
    heap "myheap" ( size = 2k );
}
```

The linker creates two labels to mark the begin and end of the heap, `_lc_ub_heap_name` for the begin of the heap and `_lc_ue_heap_name` for the end of the heap. The linker allocates space for the heap when a reference to either of the section labels exists in one of the input object files.

Reserved section

- The keyword `reserved` tells the linker to create an area or section of a given size. The linker will not locate any other sections in the memory occupied by a reserved section, with some exceptions. Each reserved section has a name. With the keyword `size` you can specify a size for a given reserved area or section.

```
group ( ... )
{
    reserved "myreserved" ( size = 2k );
}
```

The optional `fill` field contains a bit pattern that the linker writes to all memory addresses that remain unoccupied during the locate process. The result of the expression, or list of expressions, is used as values to write to memory, each in MAU. The first MAU of the fill pattern is always the first MAU in the section.

By default, no sections can overlap with a reserved section. With `alloc_allowed=absolute` sections that are located at an absolute address due to an absolute group restriction can overlap a reserved section. The same applies for reserved sections with `alloc_allowed=ranged` set. Sections restricted to a fixed address range can also overlap a reserved section.

With the `attributes` field you can set the access type of the reserved section. The linker locates the reserved section in its space with the restrictions that follow from the used attributes, `r`, `w` or `x` or a valid combination of them. The allowed attributes are shown in the following table. A value between `<` and `>` in the table means this value is set automatically by the linker.

Properties set in LSL		Resulting section properties		
attributes	filled	access	memory	content
x	yes		<rom>	executable
r	yes	r	<rom>	data
r	no	r	<rom>	scratch
rx	yes	r	<rom>	executable
rw	yes	rw	<ram>	data
rw	no	rw	<ram>	scratch
rwX	yes	rw	<ram>	executable

```
group ( ... )
{
    reserved "myreserved" ( size = 2k,
                          attributes = rw, fill = 0xaa );
}
```

If you do not specify any attributes, the linker will reserve the given number of maus, no matter what type of memory lies beneath. If you do not specify a fill pattern, no section is generated.

The linker creates two labels to mark the begin and end of the section, `_1c_ub_name` for the begin of the section and `_1c_ue_name` for the end of the reserved section.

Output sections

- The keyword `section` tells the linker to accumulate sections obtained from object files ("input sections") into an output section of a fixed size in the locate phase. You can select the input sections with `select` statements. You can use groups inside output sections, but you can only set the `align`, `attributes` and `load_addr` attributes.

The `fill` field contains a bit pattern that the linker writes to all unused space in the output section. When all input sections have an image (code/data) you must specify a fill pattern. If you do not specify a fill pattern, all input sections must be scratch sections. The fill pattern is aligned at the start of the output section.

As with a reserved section you can use the `attributes` field to set the access type of the output section.

```
group ( ... )
{
    section "myoutput" ( size = 4k, attributes = rw,
                      fill = 0xaa )
    {
        select "myinput1";
        select "myinput2";
    }
}
```

The available room for input sections is determined by the **size**, **blocksize** and **overflow** fields. With the keyword **size** you specify the fixed size of the output section. Input sections are placed from output section start towards higher addresses (offsets). When the end of the output section is reached and one or more input sections are not yet placed, an error is emitted. If however, the **overflow** field is set to another output section, remaining sections are located as if they were selected for the overflow output section.

```
group ( ... )
{
    section "tsk1_data" (size=4k, attributes=rw, fill=0,
                       overflow = "overflow_data")
    {
        select ".data.tsk1.*"
    }
    section "tsk2_data" (size=4k, attributes=rw, fill=0,
                       overflow = "overflow_data")
    {
        select ".data.tsk2.*"
    }
    section "overflow_data" (size=4k, attributes=rx,
                            fill=0)
    {
    }
}
```

With the keyword **blocksize**, the size of the output section will adapt to the size of its content. For example:

```
group flash_area (run_addr = 0x10000)
{
    section "flash_code" (blocksize=4k, attributes=rx,
                         fill=0)
    {
        select ".*.flash";
    }
}
```

If the content of the section is 1 mau, the size will be 4 kB, if the content is 11 kB, the section will be 12 kB, etc. If you use **size** in combination with **blocksize**, the **size** value is used as default (minimal) size for this section. If it is omitted, the default size will be of **blocksize**. It is not allowed to omit both **size** and **blocksize** from the section definition.

The linker creates two labels to mark the begin and end of the section, **_1c_ub_name** for the begin of the section and **_1c_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 begin of the section and `_lc_ue_table` for the end of the copy table. The linker generates a copy table when a reference to either of the section labels exists in one of the input object files.

11.8.4. Creating Symbols

You can tell the linker to create symbols before locating by putting assignments in the section layout definition. Symbol names are represented by double-quoted strings. Any string is allowed, but object files may not support all characters for symbol names. You can use two different assignment operators. With the simple assignment operator '=', the symbol is created unconditionally. With the ':=' operator, the symbol is only created if it already exists as an undefined reference in an object file.

The expression that represents the value to assign to the symbol may contain references to other symbols. If such a referred symbol is a special section symbol, creation of the symbol in the left hand side of the assignment will cause creation of the special section.

```
section_layout
{
    "_lc_cp" := "_lc_ub_table";
    // when the symbol _lc_cp occurs as an undefined reference
    // in an object file, the linker generates a copy table
}
```

11.8.5. Conditional Group Statements

Within a group, you can conditionally select sections or create special sections.

- With the `if` keyword you can specify a condition. The succeeding section statement is executed if the condition evaluates to TRUE (1).
- The optional `else` keyword is followed by a section statement which is executed in case the `if`-condition evaluates to FALSE (0).

```
group ( ... )
{
    if ( exists( "mysection" ) )
        select "mysection";
    else
        reserved "myreserved" ( size=2k );
}
```


Chapter 12. Debug Target Configuration Files

DTC files (Debug Target Configuration files) define all possible configurations for a debug target. A debug target can be target hardware such as an evaluation board or a simulator. The DTC files are used by Eclipse to configure the project and the debugger. The information is used by the Target Board Configuration wizard and the debug configuration. DTC files are located in the `etc` directory of the installed product and use `.dtc` as filename suffix.

Based on the DTC files, the Target Board Configuration wizard adjust the project's LSL file and creates a debug launch configuration.

12.1. Custom Board Support

When you need support for a custom board and the board requires a different configuration than those that are in the product, it is necessary to create a dedicated DTC file.

To add a custom board

1. From the `etc` directory of the product, make a copy of a `.dtc` file and put it in your project directory (in the current workspace).

In Eclipse, the DTC file should now be visible as part of your project.

2. Edit the file and give it a name that reflects the custom board.

The Target Board Configuration wizard in Eclipse adds DTC files that are present in your current project to the list of available target boards.

Syntax of a DTC file

DTC files are XML files. Use a delivered `.dtc` file as a starting point for creating a custom board specification.

Basically a DTC file consists of the definition of the debug target (`debugTarget` element) which embodies one or more communication methods (`communicationMethod` element). Within each communication method, you can define multiple configurations (`configuration` element). The Target Board Configuration wizard in Eclipse reflects the structure of the DTC file. The elements that determine the settings that are applied by the wizard, can be found at any level in the DTC file. The wizard will apply all elements that are within the path to the selected configuration. This is best explained by an example of a DTC file with the following basic layout:

```
debugTarget: Infineon TriBoard TC1165
  lsl
  communicationMethod: DAS over MiniWigglerII
    lsl
    configuration: Single Chip
  lsl
  communicationMethod: DAS over USB-Wiggler
    lsl
    configuration: Single Chip
```

TASKING VX-toolset for PCP User Guide

```
lsl  
lsl
```

In this example there is an LSL element at every level. If, in the Target Board Configuration wizard in Eclipse, you set the debug target configuration to "DAS over MiniWigglerII" -> "Single Chip", the wizard puts the following LSL parts into the project's LSL file in this order:

- the lsl part under the debugTarget element
- the lsl part under the communicationMethod "DAS over MiniWigglerII" element
- the lsl part under the configuration "Single Chip" in the communicationMethod "DAS over MiniWigglerII" element
- the lsl part in the debugTarget element at the end of the DTC file

The same applies to all other elements that determine the underlying settings.

DTC macros in LSL

To protect the Target Board Configuration wizard from changing the LSL file, you can protect the LSL file by adding the macro `__DTC_IGNORE`. This can be useful for projects that need the same LSL file, but still need to run on different target boards.

```
#define __DTC_IGNORE
```

The following DTC macros can be present in the LSL file:

LSL Define	Description
<code>__DTC_IGNORE</code>	If defined, protects the LSL file against changes by the Target Board Configuration wizard.
<code>__DTC_START</code> <code>__DTC_END</code>	The LSL part that is between these macros can be replaced by LSL text from the DTC file. If the macros are not present in the LSL file, the Target Board Configuration wizard will add them.

12.2. Description of DTC Elements and Attributes

The following table contains a description of the DTC elements and attributes. For each element a list of allowed elements is listed and the available attributes are described.

Element / Attribute	Description	Allowed Elements
debugTarget name manufacturer	The debug target.	flashChips, lsl, communicationMethod, def, processor, resource, initialize
	The name of the configuration.	
	The manufacturer of the debug target.	
processor	Defines a processor that can be present on the debug target. Multiple processor definitions are allowed. The user should select the actual processor on the debug target.	-

Element / Attribute	Description	Allowed Elements
name	A descriptive name of the processor derivative.	
cpu	Defines the CPU name, as for example supplied with the option <code>--cpu</code> of the C compiler.	
communicationMethod	Defines a communication method. A communication method is the channel that is used to communicate with the target.	ref, resource, initialize, configuration, lsl, processor
name	A descriptive name of the communication method.	
debugInstrument	The debug instrument DLL/Shared library file to be used for this communication method. Do not supply a path or a filename suffix.	
gdiMethod	This is the method used for communication. Allowed values: <code>rs232</code> , <code>tcpip</code> , <code>can</code> , <code>none</code>	
def	Define a set of elements as a macro. The macro can be expanded using the <code>ref</code> element.	lsl, resource, initialize, ref, configuration, flashMonitor
id	The macro name.	
resource	Defines a resource definition that can be used by Eclipse, the debugger or by the debug instrument.	-
id	The identifier name used by the debugger or debug instrument to retrieve the value.	
value	The value assigned to the resource.	
ref	Reference to a macro defined with a <code>def</code> element. The elements contained in the <code>def</code> element with the same name will be expanded at the location of the <code>ref</code> . Multiple <code>refs</code> to the same <code>def</code> are allowed.	-
id	The name of the referenced macro.	
configuration	Defines a configuration.	ref, initialize, resource, lsl, flashMonitor, processor
name	The descriptive name of the configuration.	
initialize	This element defines an initialization expression. Each initialize element contains a <code>resourceId</code> attribute. If the DI requests this resource the debugger will compose a string from all initialize elements with the same <code>resourceId</code> . This DI can use this string to initialize registers by passing it to the debugger as an expression to be evaluated.	-
resourceId	The name of the resource to be used.	

Element / Attribute	Description	Allowed Elements
name	The name of the register to be initialized.	
value	When the <code>cstart</code> attribute is false, this is the value to be used, otherwise, it is the default value when using this configuration. It will be used by the startup code editor to set the default register values.	
cstart	A boolean value. If true the debugger should ask the C startup code editor for the value, otherwise the contents of the value attribute is used. The default value is true.	
flashMonitor	This element specifies the flash programming monitor to be used for this configuration.	-
monitor	Filename of the monitor, usually an Intel Hex or S-Record file.	
workspaceAddress	The address of the workspace of the flash programming monitor.	
flashSectorBufferSize	Specifies the buffer size for buffering a flash sector.	
chip	This element defines a flash chip. It must be used by the flash properties page to add it on request to the list of flash chips.	debugTarget
vendor	The vendor of this flash chip.	
chip	The name of the chip.	
width	The width of the chip in bits.	
chips	The number of chips present on the board.	
baseAddress	The base address of the chip.	
chipSize	The size of the chip in bytes.	
flashChips	Specify a list of flash chips that can be available on this debug target.	chip
lsl	Defines LSL pieces belonging to the configuration part. The LSL text must be defined between the start and end tag of this element. All LSL texts of the active selection will be placed in the project's LSL file.	-

12.3. Special Resource Identifiers

The following resource IDs are available in the TASKING VX-toolset for TriCore:

DAS debug instrument (DI): gdi2das

Resource Name	Description	Possible Values
AccessPort	The port used to connect to the wiggler.	JTAG1, USB0
DASserver	The DAS Server used for communication.	JTAG JDRV LPT JTAG over USB Box JTAG over USB Chip
DasTimeOut	The timeout value for communication with the DAS server in milliseconds. The default is 0x4000.	
RegisterFile	The core register file that is used by the debug instrument. This is usually "regbase_f7e1.dat" or "regbase_ffff.dat", depending on the register base address.	
TerminateServer	Terminate the DAS server when the session is closed.	0, 1

12.4. Initialize Elements

The `initialize` elements are used to initialize SFRs at startup. This is also done using a resource of the debug instrument. The following resource IDs exist for the DAS debug instrument (`gdi2das`):

Resource Name	Description
<code>einit</code>	Initialize an SFR that is protected with the ENDINIT flag.
<code>init</code>	Initialize an SFR that is not protected with the ENDINIT flag.

Chapter 13. CPU Problem Bypasses and Checks

Infineon Technologies regularly publishes microcontroller errata sheets for reporting both functional problems and deviations from the electrical and timing specifications.

For some of these functional problems in the microcontroller itself, the TASKING VX-toolset for PCP provides workarounds. In fact these are software workarounds for hardware problems.

Support to deal with CPU functional problem is provided in three areas:

- Whenever possible and relevant, compiler bypasses will modify the code in order to avoid the identified erroneous code sequences;
- The assembler gives warnings for suspicious or erroneous code sequences;
- Ready-built, 'protected' standard C libraries with bypasses for all identified PCP CPU functional problems are included in the toolset.

This chapter lists a summary of functional problems which can be bypassed by the TASKING VX-toolset for PCP. Please refer to the Infineon errata sheets for the CPU step you are using, to verify if you need to use one of these bypasses.

To set a CPU bypass or check

1. From the **Project** menu, select **Properties**

The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Processor**.

In the right pane the Processor page appears.

3. From the **Processor Selection** list, select a processor.

*The **CPU Problem Bypasses and Checks** box shows the available workarounds/checks available for the selected processor.*

4. (Optional) Select **Show all CPU problem bypasses and checks**.
5. Click **Select All** or select one or more individual options.

Overview of the CPU problem bypasses and checks

The following table contains an overview of the silicon bugs you can provide to the [C compiler option --silicon-bug](#) and the [assembler option --silicon-bug](#). WA means a workaround by the compiler or assembler, CK means a check by the compiler or assembler.

TASKING VX-toolset for PCP User Guide

CPU Problem	Description	Compiler	Assembler	CPU
PCP TC.034	Usage of R7 requires delays between operations		CK	TC1767, TC1797
PCP TC.038	PCP atomic PRAM operations may operate incorrectly	WA	CK	TC1767, TC1797

PCP_TC.034

Command line option

`--silicon-bug=pcp-tc034`

Description

If the following instruction sequence is used:

instr writing to R7

directly followed by

instr reading from R7

then the second instruction will fail, providing wrong data. The write will be successful anyway.

The assembler issues a warning if the above sequence occurs.

Workaround

Add a NOP between a write to R7 followed by a read from R7.

PCP_TC.038

Command line option

`--silicon-bug=pcp-tc038`

Description

PCP atomic PRAM instructions (XCH.PI, MSET.PI, MCLR.PI) may operate incorrectly due to external FPI read-modify-write operations.

To bypass this CPU functional problem, the C compiler does not generate these atomic instructions.

The assembler issues a warning when one of the atomic PRAM instructions appear.

Chapter 14. CERT C Secure Coding Standard

The CERT C Secure Coding Standard provides rules and recommendations for secure coding in the C programming language. The goal of these rules and recommendations is to eliminate insecure coding practices and undefined behaviors that can lead to exploitable vulnerabilities. The application of the secure coding standard will lead to higher-quality systems that are robust and more resistant to attack.

This chapter contains an overview of the CERT C Secure Coding Standard recommendations and rules that are supported by the TASKING VX-toolset.

For details see the [CERT C Secure Coding Standard](#) web site. For general information about CERT secure coding, see www.cert.org/secure-coding.

Identifiers

Each rule and recommendation is given a unique identifier. These identifiers consist of three parts:

- a three-letter mnemonic representing the section of the standard
- a two-digit numeric value in the range of 00-99
- the letter "C" indicates that this is a C language guideline

The three-letter mnemonic is used to group similar coding practices and to indicate to which category a coding practice belongs.

The numeric value is used to give each coding practice a unique identifier. Numeric values in the range of 00-29 are reserved for recommendations, while values in the range of 30-99 are reserved for rules.

C compiler invocation

With the C compiler option `--cert` you can enable one or more checks for the CERT C Secure Coding Standard recommendations/rules. With `--diag=cert` you can see a list of the available checks, or you can use a three-letter mnemonic to list only the checks in a particular category. For example, `--diag=pre` lists all supported checks in the preprocessor category.

14.1. Preprocessor (PRE)

PRE01-C Use parentheses within macros around parameter names

Parenthesize all parameter names in macro definitions to avoid precedence problems.

PRE02-C Macro replacement lists should be parenthesized

Macro replacement lists should be parenthesized to protect any lower-precedence operators from the surrounding expression. The example below is syntactically correct, although the `!=` operator was omitted. Enclosing the constant `-1` in parenthesis will prevent the incorrect interpretation and force a compiler error:

```
#define EOF -1 // should be (-1)
int getchar(void);
void f(void)
{
    if (getchar() EOF) // != operator omitted
    {
        /* ... */
    }
}
```

PRE10-C Wrap multi-statement macros in a do-while loop

When multiple statements are used in a macro, enclose them in a `do-while` statement, so the macro can appear safely inside `if` clauses or other places that expect a single statement or a statement block. Braces alone will not work in all situations, as the macro expansion is typically followed by a semicolon.

PRE11-C Do not conclude a single statement macro definition with a semicolon

Macro definitions consisting of a single statement should not conclude with a semicolon. If required, the semicolon should be included following the macro expansion. Inadvertently inserting a semicolon can change the control flow of the program.

14.2. Declarations and Initialization (DCL)

DCL30-C Declare objects with appropriate storage durations

The lifetime of an automatic object ends when the function returns, which means that a pointer to the object becomes invalid.

DCL31-C Declare identifiers before using them

The ISO C90 standard allows implicit typing of variables and functions. Because implicit declarations lead to less stringent type checking, they can often introduce unexpected and erroneous behavior or even security vulnerabilities. The ISO C99 standard requires type identifiers and forbids implicit function declarations. For backwards compatibility reasons, the VX-toolset C compiler assumes an implicit declaration and continues translation after issuing a warning message (W505 or W535).

DCL32-C Guarantee that mutually visible identifiers are unique

The compiler encountered two or more identifiers that are identical in the first 31 characters. The ISO C99 standard allows a compiler to ignore characters past the first 31 in an identifier. Two distinct identifiers that are identical in the first 31 characters may lead to problems when the code is ported to a different compiler.

DCL35-C Do not invoke a function using a type that does not match the function definition

This warning is generated when a function pointer is set to refer to a function of an incompatible type. Calling this function through the function pointer will result in undefined behavior. Example:

```
void my_function(int a);
int main(void)
{
    int (*new_function)(int a) = my_function;
    return (*new_function)(10); /* the behavior is undefined */
}
```

14.3. Expressions (EXP)

EXP01-C Do not take the size of a pointer to determine the size of the pointed-to type

The size of the object(s) allocated by `malloc()`, `calloc()` or `realloc()` should be a multiple of the size of the base type of the result pointer. Therefore, the `sizeof` expression should be applied to this base type, and not to the pointer type.

EXP12-C Do not ignore values returned by functions

The compiler gives this warning when the result of a function call is ignored at some place, although it is not ignored for other calls to this function. This warning will not be issued when the function result is ignored for all calls, or when the result is explicitly ignored with a `(void)` cast.

EXP30-C Do not depend on order of evaluation between sequence points

Between two sequence points, an object should only be modified once. Otherwise the behavior is undefined.

EXP32-C Do not access a volatile object through a non-volatile reference

If an attempt is made to refer to an object defined with a volatile-qualified type through use of an lvalue with non-volatile-qualified type, the behavior is undefined.

EXP33-C Do not reference uninitialized memory

Uninitialized automatic variables default to whichever value is currently stored on the stack or in the register allocated for the variable. Consequently, uninitialized memory can cause a program to behave in an unpredictable or unplanned manner and may provide an avenue for attack.

EXP34-C Ensure a null pointer is not dereferenced

Attempting to dereference a null pointer results in undefined behavior, typically abnormal program termination.

EXP37-C Call functions with the arguments intended by the API

When a function is properly declared with function prototype information, an incorrect call will be flagged by the compiler. When there is no prototype information available at the call, the compiler cannot check the number of arguments and the types of the arguments. This message is issued to warn about this situation.

EXP38-C Do not call `offsetof()` on bit-field members or invalid types

The behavior of the `offsetof()` macro is undefined when the member designator parameter designates a bit-field.

14.4. Integers (INT)

INT30-C Ensure that unsigned integer operations do not wrap

A constant with an unsigned integer type is truncated, resulting in a wrap-around.

INT34-C Do not shift a negative number of bits or more bits than exist in the operand

The shift count of the shift operation may be negative or greater than or equal to the size of the left operand. According to the C standard, the behavior of such a shift operation is undefined. Make sure the shift count is in range by adding appropriate range checks.

INT35-C Evaluate integer expressions in a larger size before comparing or assigning to that size

If an integer expression is compared to, or assigned to a larger integer size, that integer expression should be evaluated in that larger size by explicitly casting one of the operands.

14.5. Floating Point (FLP)

FLP30-C Do not use floating point variables as loop counters

To avoid problems with limited precision and rounding, floating point variables should not be used as loop counters.

FLP35-C Take granularity into account when comparing floating point values

Floating point arithmetic in C is inexact, so floating point values should not be tested for exact equality or inequality.

FLP36-C Beware of precision loss when converting integral types to floating point

Conversion from integral types to floating point types without sufficient precision can lead to loss of precision.

14.6. Arrays (ARR)

ARR01-C Do not apply the `sizeof` operator to a pointer when taking the size of an array

A function parameter declared as an array, is converted to a pointer by the compiler. Therefore, the `sizeof` operator applied to this parameter yields the size of a pointer, and not the size of an array.

ARR34-C Ensure that array types in expressions are compatible

Using two or more incompatible arrays in an expression results in undefined behavior.

ARR35-C Do not allow loops to iterate beyond the end of an array

Reading or writing of data outside the bounds of an array may lead to incorrect program behavior or execution of arbitrary code.

14.7. Characters and Strings (STR)

STR30-C Do not attempt to modify string literals

Writing to a string literal has undefined behavior, as identical strings may be shared and/or allocated in read-only memory.

STR33-C Size wide character strings correctly

Wide character strings may be improperly sized when they are mistaken for narrow strings or for multi-byte character strings.

STR34-C Cast characters to unsigned types before converting to larger integer sizes

A signed character is sign-extended to a larger signed integer value. Use an explicit cast, or cast the value to an unsigned type first, to avoid unexpected sign-extension.

STR36-C Do not specify the bound of a character array initialized with a string literal

The compiler issues this warning when the character buffer initialized by a string literal does not provide enough room for the terminating null character.

14.8. Memory Management (MEM)

MEM00-C Allocate and free memory in the same module, at the same level of abstraction

The compiler issues this warning when the result of the call to `malloc()`, `calloc()` or `realloc()` is discarded, and therefore not `free()`d, resulting in a memory leak.

MEM08-C Use `realloc()` only to resize dynamically allocated arrays

Only use `realloc()` to resize an array. Do not use it to transform an object to an object of a different type.

MEM30-C Do not access freed memory

When memory is freed, its contents may remain intact and accessible because it is at the memory manager's discretion when to reallocate or recycle the freed chunk. The data at the freed location may appear valid. However, this can change unexpectedly, leading to unintended program behavior. As a result, it is necessary to guarantee that memory is not written to or read from once it is freed.

MEM31-C Free dynamically allocated memory exactly once

Freeing memory multiple times has similar consequences to accessing memory after it is freed. The underlying data structures that manage the heap can become corrupted. To eliminate double-free vulnerabilities, it is necessary to guarantee that dynamic memory is freed exactly once.

MEM32-C Detect and handle memory allocation errors

The result of `realloc()` is assigned to the original pointer, without checking for failure. As a result, the original block of memory is lost when `realloc()` fails.

MEM33-C Use the correct syntax for flexible array members

Use the ISO C99 syntax for flexible array members instead of an array member of size 1.

MEM34-C Only free memory allocated dynamically

Freeing memory that is not allocated dynamically can lead to corruption of the heap data structures.

MEM35-C Allocate sufficient memory for an object

The compiler issues this warning when the size of the object(s) allocated by `malloc()`, `calloc()` or `realloc()` is smaller than the size of an object pointed to by the result pointer. This may be caused by a `sizeof` expression with the wrong type or with a pointer type instead of the object type.

14.9. Environment (ENV)

ENV32-C All `atexit` handlers must return normally

The compiler issues this warning when an `atexit()` handler is calling a function that does not return. No `atexit()` registered handler should terminate in any way other than by returning.

14.10. Signals (SIG)

SIG30-C Call only asynchronous-safe functions within signal handlers

SIG32-C Do not call `longjmp()` from inside a signal handler

Invoking the `longjmp()` function from within a signal handler can lead to undefined behavior if it results in the invocation of any non-asynchronous-safe functions, likely compromising the integrity of the program.

14.11. Miscellaneous (MSC)

MSC32-C Ensure your random number generator is properly seeded

Ensure that the random number generator is properly seeded by calling `srand()`.

Chapter 15. MISRA-C Rules

This chapter contains an overview of the supported and unsupported MISRA C rules.

15.1. MISRA-C:1998

This section lists all supported and unsupported MISRA-C:1998 rules.

See also [Section 3.7.2, C Code Checking: MISRA-C](#).

A number of MISRA-C rules leave room for interpretation. Other rules can only be checked in a limited way. In such cases the implementation decisions and possible restrictions for these rules are listed.

x means that the rule is not supported by the TASKING C compiler. (R) is a required rule, (A) is an advisory rule.

1. (R) The code shall conform to standard C, without language extensions.
- x** 2. (A) Other languages should only be used with an interface standard.
3. (A) Inline assembly is only allowed in dedicated C functions.
- x** 4. (A) Provision should be made for appropriate run-time checking.
5. (R) Only use characters and escape sequences defined by ISO C.
- x** 6. (R) Character values shall be restricted to a subset of ISO 106460-1.
7. (R) Trigraphs shall not be used.
8. (R) Multibyte characters and wide string literals shall not be used.
9. (R) Comments shall not be nested.
10. (A) Sections of code should not be "commented out".

In general, it is not possible to decide whether a piece of comment is C code that is commented out, or just some pseudo code. Instead, the following heuristics are used to detect possible C code inside a comment:

- a line ends with ';', or
- a line starts with '}', possibly preceded by white space

11. (R) Identifiers shall not rely on significance of more than 31 characters.
12. (A) The same identifier shall not be used in multiple name spaces.
13. (A) Specific-length typedefs should be used instead of the basic types.
14. (R) Use `unsigned char` or `signed char` instead of plain `char`.
- x** 15. (A) Floating-point implementations should comply with a standard.
16. (R) The bit representation of floating-point numbers shall not be used.
A violation is reported when a pointer to a floating-point type is converted to a pointer to an integer type.
17. (R) `typedef` names shall not be reused.

TASKING VX-toolset for PCP User Guide

18. (A) Numeric constants should be suffixed to indicate type.
A violation is reported when the value of the constant is outside the range indicated by the suffixes, if any.
19. (R) Octal constants (other than zero) shall not be used.
20. (R) All object and function identifiers shall be declared before use.
21. (R) Identifiers shall not hide identifiers in an outer scope.
22. (A) Declarations should be at function scope where possible.
- x 23. (A) All declarations at file scope should be static where possible.
24. (R) Identifiers shall not have both internal and external linkage.
- x 25. (R) Identifiers with external linkage shall have exactly one definition.
26. (R) Multiple declarations for objects or functions shall be compatible.
- x 27. (A) External objects should not be declared in more than one file.
28. (A) The `register` storage class specifier should not be used.
29. (R) The use of a tag shall agree with its declaration.
30. (R) All automatics shall be initialized before being used .
This rule is checked using worst-case assumptions. This means that violations are reported not only for variables that are guaranteed to be uninitialized, but also for variables that are uninitialized on some execution paths.
31. (R) Braces shall be used in the initialization of arrays and structures.
32. (R) Only the first, or all enumeration constants may be initialized.
33. (R) The right hand operand of `&&` or `||` shall not contain side effects.
34. (R) The operands of a logical `&&` or `||` shall be primary expressions.
35. (R) Assignment operators shall not be used in Boolean expressions.
36. (A) Logical operators should not be confused with bitwise operators.
37. (R) Bitwise operations shall not be performed on signed integers.
38. (R) A shift count shall be between 0 and the operand width minus 1.
This violation will only be checked when the shift count evaluates to a constant value at compile time.
39. (R) The unary minus shall not be applied to an unsigned expression.
40. (A) `sizeof` should not be used on expressions with side effects.
- x 41. (A) The implementation of integer division should be documented.
42. (R) The comma operator shall only be used in a `for` condition.
43. (R) Don't use implicit conversions which may result in information loss.
44. (A) Redundant explicit casts should not be used.
45. (R) Type casting from any type to or from pointers shall not be used.
46. (R) The value of an expression shall be evaluation order independent.
This rule is checked using worst-case assumptions. This means that a violation will be reported when a possible alias may cause the result of an expression to be evaluation order dependent.

47. (A) No dependence should be placed on operator precedence rules.
48. (A) Mixed arithmetic should use explicit casting.
49. (A) Tests of a (non-Boolean) value against 0 should be made explicit.
50. (R) F.P. variables shall not be tested for exact equality or inequality.
51. (A) Constant unsigned integer expressions should not wrap-around.
52. (R) There shall be no unreachable code.
53. (R) All non-null statements shall have a side-effect.
54. (R) A null statement shall only occur on a line by itself.
55. (A) Labels should not be used.
56. (R) The `goto` statement shall not be used.
57. (R) The `continue` statement shall not be used.
58. (R) The `break` statement shall not be used (except in a `switch`).
59. (R) An `if` or loop body shall always be enclosed in braces.
60. (A) All `if, else if` constructs should contain a final `else`.
61. (R) Every non-empty `case` clause shall be terminated with a `break`.
62. (R) All `switch` statements should contain a final `default` case.
63. (A) A `switch` expression should not represent a Boolean case.
64. (R) Every `switch` shall have at least one `case`.
65. (R) Floating-point variables shall not be used as loop counters.
66. (A) A `for` should only contain expressions concerning loop control.
A violation is reported when the loop initialization or loop update expression modifies an object that is not referenced in the loop test.
67. (A) Iterator variables should not be modified in a `for` loop.
68. (R) Functions shall always be declared at file scope.
69. (R) Functions with variable number of arguments shall not be used.
70. (R) Functions shall not call themselves, either directly or indirectly.
A violation will be reported for direct or indirect recursive function calls in the source file being checked. Recursion via functions in other source files, or recursion via function pointers is not detected.
71. (R) Function prototypes shall be visible at the definition and call.
72. (R) The function prototype of the declaration shall match the definition.
73. (R) Identifiers shall be given for all prototype parameters or for none.
74. (R) Parameter identifiers shall be identical for declaration/definition.
75. (R) Every function shall have an explicit return type.
76. (R) Functions with no parameters shall have a `void` parameter list.
77. (R) An actual parameter type shall be compatible with the prototype.
78. (R) The number of actual parameters shall match the prototype.
79. (R) The values returned by `void` functions shall not be used.

TASKING VX-toolset for PCP User Guide

80. (R) Void expressions shall not be passed as function parameters.
81. (A) `const` should be used for reference parameters not modified.
82. (A) A function should have a single point of exit.
83. (R) Every exit point shall have a `return` of the declared return type.
84. (R) For `void` functions, `return` shall not have an expression.
85. (A) Function calls with no parameters should have empty parentheses.
86. (A) If a function returns error information, it should be tested.
A violation is reported when the return value of a function is ignored.
87. (R) `#include` shall only be preceded by other directives or comments.
88. (R) Non-standard characters shall not occur in `#include` directives.
89. (R) `#include` shall be followed by either `<filename>` or `"filename"`.
90. (R) Plain macros shall only be used for constants/qualifiers/specifiers.
91. (R) Macros shall not be `#define`'d and `#undef`'d within a block.
92. (A) `#undef` should not be used.
93. (A) A function should be used in preference to a function-like macro.
94. (R) A function-like macro shall not be used without all arguments.
95. (R) Macro arguments shall not contain pre-preprocessing directives.
A violation is reported when the first token of an actual macro argument is `'#'`.
96. (R) Macro definitions/parameters should be enclosed in parentheses.
97. (A) Don't use undefined identifiers in pre-processing directives.
98. (R) A macro definition shall contain at most one `#` or `##` operator.
99. (R) All uses of the `#pragma` directive shall be documented.
This rule is really a documentation issue. The compiler will flag all `#pragma` directives as violations.
100. (R) `defined` shall only be used in one of the two standard forms.
101. (A) Pointer arithmetic should not be used.
102. (A) No more than 2 levels of pointer indirection should be used.
A violation is reported when a pointer with three or more levels of indirection is declared.
103. (R) No relational operators between pointers to different objects.
In general, checking whether two pointers point to the same object is impossible. The compiler will only report a violation for a relational operation with incompatible pointer types.
104. (R) Non-constant pointers to functions shall not be used.
105. (R) Functions assigned to the same pointer shall be of identical type.
106. (R) Automatic address may not be assigned to a longer lived object.
107. (R) The null pointer shall not be de-referenced.
A violation is reported for every pointer dereference that is not guarded by a `NULL` pointer test.

- 108. (R) All `struct/union` members shall be fully specified.
- 109. (R) Overlapping variable storage shall not be used.
A violation is reported for every `union` declaration.
- 110. (R) Unions shall not be used to access the sub-parts of larger types.
A violation is reported for a `union` containing a `struct` member.
- 111. (R) Bit-fields shall have type `unsigned int` or `signed int`.
- 112. (R) Bit-fields of type `signed int` shall be at least 2 bits long.
- 113. (R) All `struct/union` members shall be named.
- 114. (R) Reserved and standard library names shall not be redefined.
- 115. (R) Standard library function names shall not be reused.
- x 116. (R) Production libraries shall comply with the MISRA C restrictions.
- x 117. (R) The validity of library function parameters shall be checked.
- 118. (R) Dynamic heap memory allocation shall not be used.
- 119. (R) The error indicator `errno` shall not be used.
- 120. (R) The macro `offsetof` shall not be used.
- 121. (R) `<locale.h>` and the `setlocale` function shall not be used.
- 122. (R) The `setjmp` and `longjmp` functions shall not be used.
- 123. (R) The signal handling facilities of `<signal.h>` shall not be used.
- 124. (R) The `<stdio.h>` library shall not be used in production code.
- 125. (R) The functions `atof/atoi/atol` shall not be used.
- 126. (R) The functions `abort/exit/getenv/system` shall not be used.
- 127. (R) The time handling functions of library `<time.h>` shall not be used.

15.2. MISRA-C:2004

This section lists all supported and unsupported MISRA-C:2004 rules.

See also [Section 3.7.2, C Code Checking: MISRA-C](#).

A number of MISRA-C rules leave room for interpretation. Other rules can only be checked in a limited way. In such cases the implementation decisions and possible restrictions for these rules are listed.

x means that the rule is not supported by the TASKING C compiler. (R) is a required rule, (A) is an advisory rule.

Environment

- 1.1 (R) All code shall conform to ISO 9899:1990 "Programming languages - C", amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996.
- 1.2 (R) No reliance shall be placed on undefined or unspecified behavior.

TASKING VX-toolset for PCP User Guide

- ✘ 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.
- ✘ 1.4 (R) The compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.
- ✘ 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

- ✘ 3.1 (R) All usage of implementation-defined behavior shall be documented.
- ✘ 3.2 (R) The character set and the corresponding encoding shall be documented.
- ✘ 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.
- ✘ 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.
- x 8.10 (R) All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.
- 8.11 (R) The `static` storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
- 8.12 (R) When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.

Initialization

- 9.1 (R) All automatic variables shall have been assigned a value before being used. This rule is checked using worst-case assumptions. This means that violations are reported not only for variables that are guaranteed to be uninitialized, but also for variables that are uninitialized on some execution paths.
- 9.2 (R) Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.
- 9.3 (R) In an enumerator list, the "=" construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.

Arithmetic type conversions

- 10.1 (R) The value of an expression of integer type shall not be implicitly converted to a different underlying type if:
 - a) it is not a conversion to a wider integer type of the same signedness, or
 - b) the expression is complex, or
 - c) the expression is not constant and is a function argument, or
 - d) the expression is not constant and is a return expression.
- 10.2 (R) The value of an expression of floating type shall not be implicitly converted to a different type if:
 - a) it is not a conversion to a wider floating type, or
 - b) the expression is complex, or
 - c) the expression is a function argument, or
 - d) the expression is a return expression.
- 10.3 (R) The value of a complex expression of integer type may only be cast to a type of the same signedness that is no wider than the underlying type of the expression.
- 10.4 (R) The value of a complex expression of floating type may only be cast to a type that is no wider than the underlying type of the expression.
- 10.5 (R) If the bitwise operators `~` and `<<` are applied to an operand of underlying type `unsigned char` or `unsigned short`, the result shall be immediately cast to the underlying type of the operand.
- 10.6 (R) A "U" suffix shall be applied to all constants of `unsigned` type.

Pointer type conversions

- 11.1 (R) Conversions shall not be performed between a pointer to a function and any type other than an integral type.
- 11.2 (R) Conversions shall not be performed between a pointer to object and any type other than an integral type, another pointer to object type or a pointer to void.
- 11.3 (A) A cast should not be performed between a pointer type and an integral type.
- 11.4 (A) A cast should not be performed between a pointer to object type and a different pointer to object type.
- 11.5 (R) A cast shall not be performed that removes any `const` or `volatile` qualification from the type addressed by a pointer.

Expressions

- 12.1 (A) Limited dependence should be placed on C's operator precedence rules in expressions.
- 12.2 (R) The value of an expression shall be the same under any order of evaluation that the standard permits. This rule is checked using worst-case assumptions. This means that a violation will be reported when a possible alias may cause the result of an expression to be evaluation order dependent.
- 12.3 (R) The `sizeof` operator shall not be used on expressions that contain side effects.
- 12.4 (R) The right-hand operand of a logical `&&` or `||` operator shall not contain side effects.
- 12.5 (R) The operands of a logical `&&` or `||` shall be *primary-expressions*.
- 12.6 (A) The operands of logical operators (`&&`, `||` and `!`) should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to operators other than (`&&`, `||` and `!`).
- 12.7 (R) Bitwise operators shall not be applied to operands whose underlying type is signed.
- 12.8 (R) The right-hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left-hand operand. This violation will only be checked when the shift count evaluates to a constant value at compile time.
- 12.9 (R) The unary minus operator shall not be applied to an expression whose underlying type is unsigned.
- 12.10 (R) The comma operator shall not be used.
- 12.11 (A) Evaluation of constant unsigned integer expressions should not lead to wrap-around.
- 12.12 (R) The underlying bit representations of floating-point values shall not be used. A violation is reported when a pointer to a floating-point type is converted to a pointer to an integer type.
- 12.13 (A) The increment (`++`) and decrement (`--`) operators should not be mixed with other operators in an expression.

Control statement expressions

- 13.1 (R) Assignment operators shall not be used in expressions that yield a Boolean value.
- 13.2 (A) Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.
- 13.3 (R) Floating-point expressions shall not be tested for equality or inequality.
- 13.4 (R) The controlling expression of a `for` statement shall not contain any objects of floating type.
- 13.5 (R) The three expressions of a `for` statement shall be concerned only with loop control. A violation is reported when the loop initialization or loop update expression modifies an object that is not referenced in the loop test.
- 13.6 (R) Numeric variables being used within a `for` loop for iteration counting shall not be modified in the body of the loop.
- 13.7 (R) Boolean operations whose results are invariant shall not be permitted.

Control flow

- 14.1 (R) There shall be no unreachable code.
- 14.2 (R) All non-null statements shall either:
 - a) have at least one side effect however executed, or
 - b) cause control flow to change.
- 14.3 (R) Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment provided that the first character following the null statement is a white-space character.
- 14.4 (R) The `goto` statement shall not be used.
- 14.5 (R) The `continue` statement shall not be used.
- 14.6 (R) For any iteration statement there shall be at most one `break` statement used for loop termination.
- 14.7 (R) A function shall have a single point of exit at the end of the function.
- 14.8 (R) The statement forming the body of a `switch`, `while`, `do ... while` or `for` statement be a compound statement.
- 14.9 (R) An `if (expression)` construct shall be followed by a compound statement. The `else` keyword shall be followed by either a compound statement, or another `if` statement.
- 14.10 (R) All `if ... else if` constructs shall be terminated with an `else` clause.

Switch statements

- 15.1 (R) A switch label shall only be used when the most closely-enclosing compound statement is the body of a `switch` statement.
- 15.2 (R) An unconditional `break` statement shall terminate every non-empty `switch` clause.
- 15.3 (R) The final clause of a `switch` statement shall be the `default` clause.
- 15.4 (R) A `switch` expression shall not represent a value that is effectively Boolean.
- 15.5 (R) Every `switch` statement shall have at least one `case` clause.

Functions

- 16.1 (R) Functions shall not be defined with variable numbers of arguments.
- 16.2 (R) Functions shall not call themselves, either directly or indirectly. A violation will be reported for direct or indirect recursive function calls in the source file being checked. Recursion via functions in other source files, or recursion via function pointers is not detected.
- 16.3 (R) Identifiers shall be given for all of the parameters in a function prototype declaration.
- 16.4 (R) The identifiers used in the declaration and definition of a function shall be identical.
- 16.5 (R) Functions with no parameters shall be declared with parameter type `void`.
- 16.6 (R) The number of arguments passed to a function shall match the number of parameters.

- 16.7 (A) A pointer parameter in a function prototype should be declared as pointer to `const` if the pointer is not used to modify the addressed object.
- 16.8 (R) All exit paths from a function with non-void return type shall have an explicit `return` statement with an expression.
- 16.9 (R) A function identifier shall only be used with either a preceding `&`, or with a parenthesized parameter list, which may be empty.
- 16.10 (R) If a function returns error information, then that error information shall be tested. A violation is reported when the return value of a function is ignored.

Pointers and arrays

- x 17.1 (R) Pointer arithmetic shall only be applied to pointers that address an array or array element.
- x 17.2 (R) Pointer subtraction shall only be applied to pointers that address elements of the same array.
- 17.3 (R) `>`, `>=`, `<`, `<=` shall not be applied to pointer types except where they point to the same array. In general, checking whether two pointers point to the same object is impossible. The compiler will only report a violation for a relational operation with incompatible pointer types.
- 17.4 (R) Array indexing shall be the only allowed form of pointer arithmetic.
- 17.5 (A) The declaration of objects should contain no more than 2 levels of pointer indirection. A violation is reported when a pointer with three or more levels of indirection is declared.
- 17.6 (R) The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

Structures and unions

- 18.1 (R) All structure or union types shall be complete at the end of a translation unit.
- 18.2 (R) An object shall not be assigned to an overlapping object.
- x 18.3 (R) An area of memory shall not be reused for unrelated purposes.
- 18.4 (R) Unions shall not be used.

Preprocessing directives

- 19.1 (A) `#include` statements in a file should only be preceded by other preprocessor directives or comments.
- 19.2 (A) Non-standard characters should not occur in header file names in `#include` directives.
- x 19.3 (R) The `#include` directive shall be followed by either a `<filename>` or `"filename"` sequence.
- 19.4 (R) C macros shall only expand to a braced initializer, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct.

TASKING VX-toolset for PCP User Guide

- 19.5 (R) Macros shall not be `#define`'d or `#undef`'d within a block.
- 19.6 (R) `#undef` shall not be used.
- 19.7 (A) A function should be used in preference to a function-like macro.
- 19.8 (R) A function-like macro shall not be invoked without all of its arguments.
- 19.9 (R) Arguments to a function-like macro shall not contain tokens that look like preprocessing directives. A violation is reported when the first token of an actual macro argument is `'#'`.
- 19.10 (R) In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of `#` or `##`.
- 19.11 (R) All macro identifiers in preprocessor directives shall be defined before use, except in `#ifdef` and `#ifndef` preprocessor directives and the `defined()` operator.
- 19.12 (R) There shall be at most one occurrence of the `#` or `##` preprocessor operators in a single macro definition.
- 19.13 (A) The `#` and `##` preprocessor operators should not be used.
- 19.14 (R) The `defined` preprocessor operator shall only be used in one of the two standard forms.
- 19.15 (R) Precautions shall be taken in order to prevent the contents of a header file being included twice.
- 19.16 (R) Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.
- 19.17 (R) All `#else`, `#elif` and `#endif` preprocessor directives shall reside in the same file as the `#if` or `#ifdef` directive to which they are related.

Standard libraries

- 20.1 (R) Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined.
- 20.2 (R) The names of standard library macros, objects and functions shall not be reused.
- x 20.3 (R) The validity of values passed to library functions shall be checked.
- 20.4 (R) Dynamic heap memory allocation shall not be used.
- 20.5 (R) The error indicator `errno` shall not be used.
- 20.6 (R) The macro `offsetof`, in library `<stddef.h>`, shall not be used.
- 20.7 (R) The `setjmp` macro and the `longjmp` function shall not be used.
- 20.8 (R) The signal handling facilities of `<signal.h>` shall not be used.
- 20.9 (R) The input/output library `<stdio.h>` shall not be used in production code.
- 20.10 (R) The library functions `atof`, `atoi` and `atol` from library `<stdlib.h>` shall not be used.
- 20.11 (R) The library functions `abort`, `exit`, `getenv` and `system` from library `<stdlib.h>` shall not be used.
- 20.12 (R) The time handling functions of library `<time.h>` shall not be used.

Run-time failures

- ✘ 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.

