MA160-025-00-00
Doc. ver.: 1.1

PCP v2.5

C Compiler, Assembler, Linker User's Manual

TASKING
Embedded software development from Altium

A publication of

Altium BV

Documentation Department

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

CONTENTS

TABLE OF CONTENTS

TASKING

CONTENTS

USING THE COMPILER 4-1

USING THE ASSEMBLER 5-1

USING THE LINKER 6-1

MANUAL PURPOSE AND STRUCTURE

Windows Users

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

UNIX Users

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

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

Structure

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

First you need to install the software. This is described in Chapter 1, *Software Installation and Configuration*

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

Once you are familiar with these tools, you can use the Reference Manual to lookup specific options and details to make full use of the PCP toolchain.

SHORT TABLE OF CONTENTS

CONVENTIONS USED IN THIS MANUAL

Notation for syntax

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

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

Example

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

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

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

Not valid is:

```
cpcp -g
```

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

Icons

The following illustrations are used in this manual:

Note: notes give you extra information.

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

Command line: type your input on the command line.

Reference: follow this reference to find related topics.

RELATED PUBLICATIONS

C Standards

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

MISRA-C

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

TASKING Tools

- PCP C Compiler, Assembler, Linker Reference Manual
  [Altium, MB160-025-00-00]

MANUAL STRUCTURE

CHAPTER 1

SOFTWARE INSTALLATION AND CONFIGURATION

TASKING

1 CHAPTER

1.1 INTRODUCTION

This chapter guides you through the procedures to install the software on a Windows system or on a Linux or UNIX host.

After the installation, it is explained how to configure the software and how to install the license information that is needed to actually use the software.

1.2 SOFTWARE INSTALLATION

1.2.1 INSTALLATION FOR WINDOWS

1. Start Windows 95/98/XP/NT/2000, if you have not already done so.
2. Insert the CD-ROM into the CD-ROM drive.

   If the TASKING Showroom dialog box appears, proceed with Step 5.
3. Click the **Start** button and select **Run...**
4. In the dialog box type **`d:\setup`** (substitute the correct drive letter for your CD-ROM drive) and click on the **OK** button.

   The TASKING Showroom dialog box appears.
5. Select a product and click on the **Install** button.
6. Follow the instructions that appear on your screen.

You can find your serial number on the invoice, delivery note, or picking slip delivered with the product.

7. License the software product as explained in section 1.4, *Licensing TASKING Products*.

1.2.2 INSTALLATION FOR LINUX

Each product on the CD-ROM is available as an RPM package, Debian package and as a gzipped tar file. For each product the following files are present:

```
SWproduct-version-RPMrelease.i386.rpm
swproduct_version-release_i386.deb
SWproduct-version.tar.gz
```

These three files contain exactly the same information, so you only have to install one of them. When your Linux distribution supports RPM packages, you can install the `.rpm` file. For a Debian based distribution, you can use the `.deb` file. Otherwise, you can install the product from the `.tar.gz` file.

RPM Installation

1. In most situations you have to be "root" to install RPM packages, so either login as "root", or use the **su** command.

2. Insert the CD-ROM into the CD-ROM drive. Mount the CD-ROM on a directory, for example `/cdrom`. See the Linux manual pages about **mount** for details.

3. Go to the directory on which the CD-ROM is mounted:

   ```
   cd /cdrom
   ```

4. To install or upgrade all products at once, issue the following command:

   ```
   rpm -U SW*.rpm
   ```

   This will install or upgrade all products in the default installation directory `/usr/local`. Every RPM package will create a single directory in the installation directory.

   The RPM packages are 'relocatable', so it is possible to select a different installation directory with the **--prefix** option. For instance when you want to install the products in `/opt`, use the following command:

   ```
   rpm -U --prefix /opt SW*.rpm
   ```

For Red Hat 6.0 users: The **--prefix** option does not work with RPM version 3.0, included in the Red Hat 6.0 distribution. Please upgrade to RPM verion 3.0.3 or higher, or use the `.tar.gz` file installation described in the next section if you want to install in a non-standard directory.

Debian Installation

1. Login as a user.

   Be sure you have read, write and execute permissions in the installation directory. Otherwise, login as ”root” or use the **su** command.

2. Insert the CD-ROM into the CD-ROM drive. Mount the CD-ROM on a directory, for example `/cdrom`. See the Linux manual pages about **mount** for details.

3. Go to the directory on which the CD-ROM is mounted:

   ```
   cd /cdrom
   ```

4. To install or upgrade all products at once, issue the following command:

   ```
   dpkg -i sw*.deb
   ```

   This will install or upgrade all products in a subdirectory of the default installation directory `/usr/local`.

Tar.gz Installation

1. Login as a user.

   Be sure you have read, write and execute permissions in the installation directory. Otherwise, login as ”root” or use the **su** command.

2. Insert the CD-ROM into the CD-ROM drive. Mount the CD-ROM on a directory, for example `/cdrom`. See the Linux manual pages about **mount** for details.

3. Go to the directory on which the CD-ROM is mounted:

   ```
   cd /cdrom
   ```

4. To install the products from the `.tar.gz` files in the directory `/usr/local`, issue the following command for each product:

   ```
   tar xzf SWproduct-version.tar.gz -C /usr/local
   ```

   Every `.tar.gz` file creates a single directory in the directory where it is extracted.

1.2.3 INSTALLATION FOR UNIX HOSTS

1. Login as a user.

   Be sure you have read, write and execute permissions in the installation directory. Otherwise, login as "root" or use the **su** command.

   If you are a first time user, decide where you want to install the product. By default it will be installed in `/usr/local`.

2. Insert the CD-ROM into the CD-ROM drive and mount the CD-ROM on a directory, for example `/cdrom`.

   Be sure to use an ISO 9660 file system with Rock Ridge extensions enabled. See the UNIX manual pages about **mount** for details.

3. Go to the directory on which the CD-ROM is mounted:

   ```
   cd /cdrom
   ```

4. Run the installation script:

   ```
   sh install
   ```

   Follow the instructions appearing on your screen.

   First a question appears about where to install the software. The default answer is **`/usr/local`**.

   On some hosts the installation script asks if you want to install SW000098, the Flexible License Manager (FLEXlm). If you do not already have FLEXlm on your system, you must install it otherwise the product will not work on those hosts. See section 1.4, *Licensing TASKING Products*.

   If the script detects that the software has been installed before, the following messages appear on the screen:

   ```
           *** WARNING ***
   SWxxxxxx xxxx.xxxx already installed.
   Do you want to REINSTALL? [y,n]
   ```

   Answering **n** (no) to this question causes installation to abort and the following message being displayed:

   ```
   => Installation stopped on user request <=
   ```

Answer **y** (yes) to continue with the installation. The last message will be:

```
Installation of SWxxxxxx xxxx.xxxx completed.
```

5. If you purchased a protected TASKING product, license the software product as explained in section 1.4, *Licensing TASKING Products*.

1.3 SOFTWARE CONFIGURATION

Now you have installed the software, you can configure both the Embedded Development Environment and the command line environment for Windows, Linux and UNIX.

1.3.1 CONFIGURING THE COMMAND LINE ENVIRONMENT

To facilitate the invocation of the tools from the command line (either using a Windows command prompt or using Linux or UNIX), you can set *environment variables*.

You can set the following variables:

Environment Variable	Description
PATH	With this variable you specify the directory in which the executables reside (for example: `c:\cpcp\bin`). This allows you to call the executables when you are not in the `bin` directory. Usually your system already uses the PATH variable for other purposes. To keep these settings, you need to add (rather than replace) the path. Use a semicolon (;) to separate pathnames.
CPCPINC	With this variable you specify one or more additional directories in which the C compiler **cpcp** looks for include files. The compiler first looks in these directories, then always looks in the default `include` directory relative to the installation directory.
ASPCPINC	With this variable you specify one or more additional directories in which the assembler **aspcp** looks for include files. The assembler first looks in these directories, then always looks in the default `include` directory relative to the installation directory.
CCPCPBIN	With this variable you specify the directory in which the control program **ccpcp** looks for the executable tools. The path you specify here should match the path that you specified for the PATH variable.
CCPCPOPT	With this variable you specify options and/or arguments to each invocation of the control program **ccpcp**. The control program processes these arguments before the command line arguments.
LM_LICENSE_FILE	With this variable you specify the location of the license data file. You only need to specify this variable if the license file is not on its default location (`c:\flexlm` for Windows, `/usr/local/flexlm/licenses` for UNIX).

Environment Variable	Description
TASKING_LIC_WAIT	If you set this variable, the tool will wait for a license to become available, if all licenses are taken. If you have not set this variable, the tool aborts with an error message. (Only useful with floating licenses)
TMPDIR	With this variable you specify the location where programs can create temporary files. Usually your system already uses this variable. In this case you do not need to change it.

Table 1-1: Environment variables

The following examples show how to set an environment variable using the PATH variable as an example.

Example for Windows 95/98

Add the following line to your `autoexec.bat` file:

```
set PATH=%path%;c:\cpcp\bin
```

You can also type this line in a Command Prompt window but you will loose this setting after you close the window.

Example for Windows NT

1. Right-click on the `My Computer` icon on your desktop and select **Properties** from the menu.

   The System Properties dialog appears.

2. Select the **Environment** tab.

3. In the list of **System Variables** select **Path**.

4. In the **Value** field, add the path where the executables are located to the existing path information. Separate pathnames with a semicolon (;). For example: `c:\cpcp\bin`.

5. Click on the **Set** button, then click **OK**.

Example for Windows XP / 2000

1. Right-click on the `My Computer` icon on your desktop and select **Properties** from the menu.

   The System Properties dialog appears.

2. Select the **Advanced** tab.

3. Click on the **Environment Variables** button.

   The Environment Variables dialog appears.

4. In the list of **System variables** select **Path**.

5. Click on the **Edit** button.

   The Edit System Variable dialog appears.

6. In the **Variable value** field, add the path where the executables are located to the existing path information. Separate pathnames with a semicolon (;). For example: `c:\cpcp\bin`.

7. Click on the **OK** button to accept the changes and close the dialogs.

Example for UNIX

Enter the following line (C-shell):

```
setenv PATH $PATH:/usr/local/cpcp/bin
```

1.4 LICENSING TASKING PRODUCTS

TASKING products are protected with license management software (FLEXlm). To use a TASKING product, you must install the license key provided by TASKING for the type of license purchased.

You can run TASKING products with a node-locked license or with a floating license. When you order a TASKING product determine which type of license you need (UNIX products only have a floating license).

Node-locked license (PC only)

This license type locks the software to one specific PC so you can use the product on that particular PC only.

Floating license

This license type manages the use of TASKING product licenses among users at one site. This license type does not lock the software to one specific PC or workstation but it requires a network. The software can then be used on any computer in the network. The license specifies the number of users who can use the software simultaneously. A system allocating floating licenses is called a **license server**. A license manager running on the license server keeps track of the number of users.

1.4.1 OBTAINING LICENSE INFORMATION

Before you can install a software license you must have a "License Key" containing the license information for your software product. If you have not received such a license key follow the steps below to obtain one. Otherwise, you can install the license.

Windows

1. Run the License Administrator during installation and follow the steps to **Request a license key from Altium by E-mail**.
2. E-mail the license request to your local TASKING sales representative. The license key will be sent to you by E-mail.

UNIX

1. If you need a floating license on UNIX, you must determine the host ID and host name of the computer where you want to use the license manager. Also decide how many users will be using the product. See section 1.4.5, *How to Determine the Host ID* and section 1.4.6, *How to Determine the Host Name*.

2. When you order a TASKING product, provide the host ID, host name and number of users to your local TASKING sales representative. The license key will be sent to you by E-mail.

1.4.2 INSTALLING NODE-LOCKED LICENSES

If you do not have received your license key, read section 1.4.1, *Obtaining License Information*, before continuing.

1. Install the TASKING software product following the installation procedure described in section 1.2.1, *Installation for Windows*, if you have not done this already.

2. Create a license file by importing a license key or create one manually:

Import a license key

During installation you will be asked to run the License Administrator. Otherwise, start the License Administrator (**licadmin.exe**) manually.

In the License Administrator follow the steps to **Import a license key received from Altium by E-mail**. The License Administrator creates a license file for you.

Create a license file manually

If you prefer to create a license file manually, create a file called "`license.dat`" in the `c:\flexlm` directory, using an ASCII editor and insert the license key information received by E-mail in this file. This file is called the "license file". If the directory `c:\flexlm` does not exist, create the directory.

If you wish to install the license file in a different directory, see section 1.4.4, *Modifying the License File Location*.

If you already have a license file, add the license key information to the existing license file. If the license file already contains any SERVER lines, you must use another license file. See section 1.4.4, *Modifying the License File Location*, for additional information.

The software product and license file are now properly installed.

1.4.3 INSTALLING FLOATING LICENSES

If you do not have received your license key, read section 1.4.1, *Obtaining License Information*, before continuing.

1. Install the TASKING software product following the installation procedure described earlier in this chapter on each computer or workstation where you will use the software product.
2. On each PC or workstation where you will use the TASKING software product the location of a license file must be known, containing the information of all licenses. Either create a local license file or point to a license file on a server:

Add a licence key to a local license file

A local license file can reduce network traffic.

On Windows, you can follow the same steps to import a license key or create a license file manually, as explained in the previous section with the installation of a node-locked license.

On UNIX, you have to insert the license key manually in the license file. The default location of the license file `license.dat` is in directory `/usr/local/flexlm/licenses` for UNIX.

If you wish to install the license file in a different directory, see section 1.4.4, *Modifying the License File Location*.

If you already have a license file, add the license key information to the existing license file. If the license file already contains any SERVER lines, make sure that the number of SERVER lines and their contents match, otherwise you must use another license file. See section 1.4.4, *Modifying the License File Location*, for additional information.

Point to a license file on the server

Set the environment variable **LM_LICENSE_FILE** to "*port@host*", where *host* and *port* come from the SERVER line in the license file. On Windows, you can use the License Administrator to do this for you. In the License Administrator follow the steps to **Point to a FLEXlm License Server to get your licenses**.

3. If you already have installed FLEXlm v8.4 or higher (for example as part of another product) you can skip this step and continue with step 4. Otherwise, install SW000098, the Flexible License Manager (FLEXlm), on the license server where you want to use the license manager.

It is not recommended to run a license manager on a Windows 95 or Windows 98 machine. Use Windows XP, NT or 2000 instead, or use UNIX or Linux.

4. If FLEXlm has already been installed as part of a non-TASKING product you have to make sure that the `bin` directory of the FLEXlm product contains a copy of the **Tasking** daemon. This file part of the TASKING product installation and is present in the `flexlm` subdirectory of the toolchain. This file is also on every product CD that includes FLEXlm, in directory `licensing`.

5. On the license server also add the license key to the license file. Follow the same instructions as with "Add a license key to a local license file" in step 2.

See the FLEXlm PDF manual delivered with SW000098, which is present on each TASKING product CD, for more information.

1.4.4 MODIFYING THE LICENSE FILE LOCATION

The default location for the license file on Windows is:

```
c:\flexlm\license.dat
```

On UNIX this is:

```
/usr/local/flexlm/licenses/license.dat
```

If you want to use another name or directory for the license file, each user must define the environment variable **LM_LICENSE_FILE**.

If you have more than one product using the FLEXlm license manager you can specify multiple license files to the **LM_LICENSE_FILE** environment variable by separating each pathname (*lfpath*) with a '**;**' (on UNIX '**:**'):

Example Windows:

```
set LM_LICENSE_FILE=c:\flexlm\license.dat;c:\license.txt
```

Example UNIX:

```
setenv LM_LICENSE_FILE
/usr/local/flexlm/licenses/license.dat:/myprod/license.txt
```

If the license file is not available on these hosts, you must set **LM_LICENSE_FILE** to *port@host*; where *host* is the host name of the system which runs the FLEXlm license manager and *port* is the TCP/IP port number on which the license manager listens.

To obtain the port number, look in the license file at *host* for a line starting with "SERVER". The fourth field on this line specifies the TCP/IP port number on which the license server listens. For example:

```
setenv LM_LICENSE_FILE 7594@elliot
```

See the FLEXlm PDF manual delivered with SW000098, which is present on each TASKING product CD, for detailed information.

1.4.5 HOW TO DETERMINE THE HOST ID

The host ID depends on the platform of the machine. Please use one of the methods listed below to determine the host ID.

Platform	Tool to retrieve host ID	Example host ID
HP-UX	**lanscan** (use the station address without the leading '0x')	0000F0050185
Linux	**hostid**	11ac5702
SunOS/Solaris	**hostid**	170a3472
Windows	**licadmin** (License Administrator, or use **lmhostid**)	0060084dfbe9

Table 1-2: Determine the host ID

On Windows, the License Administrator (**licadmin**) helps you in the process of obtaining your license key.

If you do not have the program **licadmin** you can download it from our Web site at: http://www.tasking.com/support/flexlm/licadmin.zip . It is also on every product CD that includes FLEXlm, in directory `licensing`.

1.4.6 HOW TO DETERMINE THE HOST NAME

To retrieve the host name of a machine, use one of the following methods.

Platform	Method
UNIX	**hostname**
Windows NT	**licadmin** or: Go to the Control Panel, open "Network". In the "Identification" tab look for "Computer Name".
Windows XP/2000	**licadmin** or: Go to the Control Panel, open "System". In the "Computer Name" tab look for "Full computer name".

Table 1-3: Determine the host name

CHAPTER 2

PCP C LANGUAGE

TASKING

2 CHAPTER

2.1 INTRODUCTION

The TASKING PCP cross-compiler (**cpcp**) fully supports the ISO C standard and adds extra possibilities to program the special functions of the PCP.

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

- intrinsic (built-in) functions that result in PCP 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 to specify memory types for data and functions
- attributes to specify alignment and absolute addresses

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

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

2.2 DATA TYPES

The PCP architecture defines the following fundamental data types:

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

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

Type	Keyword	Size (bit)	Align (bit)	Ranges
	`unsigned int` `unsigned long`	32	32	$0 .. 2^{32}-1$
	`enum`	32	32	$-2^{31} .. 2^{31}-1$
	`long long` `signed long long`	32	32	$-2^{31} .. -2^{31}-1$
	`unsigned long long`	32	32	$0 .. 2^{32}-1$
Pointer	pointer to data pointer to func	32	32	$0 .. 2^{31}-1$ $0 .. 2^{32}-1$
Floating-Point	`float`	32	32	$-3.402e^{38} .. -1.175e^{-38}$ $1.175e^{-38} .. 3.402^{e38}$
	`double` `long double`	32	32	$-3.402e^{38} .. -1.175e^{-38}$ $1.175e^{-38} .. 3.402^{e38}$

Table 2-1: Fundamental Data Types

Aggregate types are aligned on 32 bit by default. All members of the aggregate types are aligned as required by their individual types as listed in Fundamental Data Types. This may result in gaps caused by internal padding. In addition, the total size of a struct/union must be equal to an integral multiple of its alignment. Therefore, tail padding must be applied as necessary. 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 for 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, prevents any other bit-field from residing in the storage unit corresponding to the type of the zero-width bit-field.

2.3 MEMORY QUALIFIERS

2.3.1 DECLARE A DATA OBJECT AT AN ABSOLUTE ADDRESS: __at()

You can place an object at an *absolute address* in memory. This may be useful to interface with other programs using fixed memory schemes, or to access special function registers.

With the attribute `__at()` you can specify an absolute address.

Examples

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

The variable `myvar` is placed at address 0x100.

```
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`.

Restrictions

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

- You can place only global variables at absolute addresses. Parameters, or automatic variables within functions cannot be placed at absolute addresses.
- Variables that are declared `extern`, cannot be placed at absolute addresses.
- When the variable is declared `static`, no public symbol will be generated (normal C behavior).
- You cannot place functions at absolute addresses.

- 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.
- When you declare the same absolute variable within two modules, this produces conflicts during link time (except when one of the modules declares the variable 'extern').

2.4 INTRINSIC FUNCTIONS

Some specific PCP 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 then generates the most efficient assembly code for these functions.

The compiler always inlines the corresponding assembly instructions in the assembly source rather than calling the function. This avoids unnecessary parameter passing and register saving instructions which are normally necessary when a function is called.

Intrinsic functions produce very efficient assembly code. Though it is possible to inline assembly code by hand, registers are used even more efficient by intrinsic functions. 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
```

You can use the following intrinsic functions:

__alloc()

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

Allocate memory. Same as library function `malloc()`. Returns a pointer to space in external memory of size bytes length. NULL if there is not enough space left.

__dotdotdot__()

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

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

__free()

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

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

__nop()

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

Inserts a NOP instruction.

__get_return_address()

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

Used by the compiler in `retjmp()`.

__ld32_fpi()

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

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

Example:

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

generates:

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

__st32_fpi()

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

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

Example:

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

generates:

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

__exit()

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

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

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

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

2.5 USING ASSEMBLY IN THE C SOURCE: __asm()

With the **__asm()** keyword 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. 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]]] );
```

instruction_template — Assembly instructions that may contain parameters from the input list or output list in the form: **%***parm_nr*

%*parm_nr* — Parameter number in the range 0 .. 7. With the optional ***.regnum*** you can access an individual register from a register pair or register quad.

output_param_list — [[**"=**[**&**]*constraint_char***"(***C_expression***)**],...]

input_param_list — [[**"***constraint_char***"(***C_expression***)**],...]

& — Says that an output operand is written to before the inputs are read, so this output must not be the same register as any input.

constraint _char — Constraint character: the type of register to be used for the *C_expression*. (see table 2-2)

C_expression — Any C expression. For output parameters it must be an *lvalue*, that is, something that is legal to have on the left side of an assignment.

register_save_list — [[**"***register_name***"**],...]

register_name — Name of the register you want to reserve.

Typical example: multiplying two C variables using assembly

```
int a,b,result;

void main( 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" : "=w"(result) : "w"(a), "w"(b) );
}
```

generated code:

```
ldl.il  r7,@DPTR(_PCP_a)
ld.pi   r5,[_PCP_a]
ld.pi   r1,[_PCP_b]
minit   r5,r1
mstep.u r5,r1
mstep.u r5,r1
mstep.u r5,r1
mstep.u r5,r1
ldl.il  r7,@DPTR(_PCP_result)
st.pi   r5,[_PCP_result]
jc.ia   r2,cc_uc
```

`%0` corresponds to the first C variable, `%1` corresponds to the second and so on. The escape sequence `\t` generates a tab.

Specifying registers for C variables

With a *constraint character* you specify the register *type* for a parameter. In the example above, the `w` is used to force the use of word registers for the parameters `a`, `b` and `result`.

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

Table 2-2: Available input/output operand constraints

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.

Example 1: no input or output

A simple example without input or output parameters. You can just output any assembly instruction:

```
__asm( "nop" );
```

Generated code:

```
nop
```

Example 2: using output parameters

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

```
int result;

void main( void )
{
    __asm( "mov %0,#0xFF" : "=w"(result));
}
```

Generated assembly code:

```
mov    r5,#0xFF
ldl.il r7,@DPTR(_PCP_result)
st.pi  r5,[_PCP_result]
jc.ia  r2,cc_uc
```

Example 3: using input and output parameters

Multiply two C variables and assign the result to a third C variable. Word registers are necessary for the input and output parameters (constraint `w`, `%0` for `result`, `%1` for `a` and `%2` for `b` 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 a, b, result;

void mymul( 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" : "=w"(result) : "w"(a), "w"(b) );
}

void main(void)
{
    myfunc();
}
```

Generated assembly code:

```
_PCP_myfunc:    .type   func
        ldl.il  r7,@DPTR(_PCP_a)
        ld.pi   r5,[_PCP_a]
        ld.pi   r1,[_PCP_b]
        minit   r5,r1
        mstep.u r5,r1
        mstep.u r5,r1
        mstep.u r5,r1
        mstep.u r5,r1
        ldl.il  r7,@DPTR(_PCP_result)
        st.pi   r5,[_PCP_result]
        jc.ia   r2,cc_uc
```

Example 4: reserve registers

If you use registers in the **__asm** statement, reserve them. Same as *Example 3*, but now register **r1** is a reserved register. You can do this by adding a reserved register list (**: "r5"**) (sometimes referred to as 'clobber list'). As you can see in the generated assembly code, register **r5** is not used (**r3** is used instead).

```
int a, b, result;

void mymul( 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":"=w"(result) : "w"(a), "w"(b) : "r5" );
}
```

Generated assembly code:

```
_PCP_myfunc:    .type   func
        ldl.il  r7,@DPTR(_PCP_a)
        ld.pi   r1,[_PCP_a]
        ld.pi   r3,[_PCP_b]
        minit   r1,r3
        mstep.u r1,r3
        mstep.u r1,r3
        mstep.u r1,r3
        mstep.u r1,r3
        ldl.il  r7,@DPTR(_PCP_result)
        st.pi   r1,[_PCP_result]
        jc.ia   r2,cc_uc
```

Example 5: 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 **ivar** and returns this value to **ovar**. Since the assembly instruction **not** uses only one register, the return value has to go in the same place as the input value. To indicate that **ivar** uses the same register as **ovar**, the constraint '0' is used which indicates that **ivar** also corresponds with %0.

```
int ivar,ovar;

static inline void invert(int ivar)
/* 'static inline' makes assembly easier to read */
{
    __asm ("not %0,%0": "=w"(ovar): "0"(ivar) );
}

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

Generated assembly code:

```
ldl.il  r7,@DPTR(_PCP_ivar)
ld.pi   r5,[_PCP_ivar]
not r5,r5
ldl.il  r7,@DPTR(_PCP_ovar)
st.pi   r5,[_PCP_ovar]
jc.ia   r2,cc_uc
```

Example 6: 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.

First write a function with assembly in the body using the keyword **__asm**. We use the multiply routine from *Example 3*.

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 2.9.1, *Inlining Functions*.

```
int a, b, result;

inline void __mymul( 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" : "=w"(result) : "w"(a), "w"(b) );
}

void main(void)
{
    // call to function __mymul
    __mymul();
}
```

Generated assembly code:

```
_PCP_main:      .type   func
        ldl.il  r7,@DPTR(_PCP_a)
        ld.pi   r5,[_PCP_a]
        ld.pi   r1,[_PCP_b]
        minit   r5,r1
        mstep.u r5,r1
        mstep.u r5,r1
        mstep.u r5,r1
        mstep.u r5,r1
        ldl.il  r7,@DPTR(_PCP_result)
        st.pi   r5,[_PCP_result]
        jc.ia   r2,cc_uc
```

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

2.6 CONTROLLING THE COMPILER: PRAGMAS

Pragmas are keywords in the C source that control the behavior of the compiler. Pragmas sometimes overrule compiler options. In general pragmas give directions to the code generator of the compiler.

The syntax is:

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

or:

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

For example, you can set a compiler option to specify which optimizations the compiler should perform. With the `#pragma optimize flags` you can set an optimization level for a specific part of the C source. This overrules the general optimization level that is set in the compiler options dialog (command line option **-O**).

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.

See section 5.1, *Compiler Options*, in Chapter 5, *Tool Options*, of the *Reference Manual*.

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

Pragma name	Description
`alias symbol=defined-symbol`	Defines an alias for a symbol
`clear` `noclear`	Specifies 'clearing' of non-initialized static/public variables
`compactmaxmatch value`	Controls the maximum size of a match.
`extension isuffix`	Enables the language extension to specify imaginary floating-point constants by adding an 'i' to the constant
`extern symbol`	Forces an external reference

`inline`
`noinline`
`smartinline`
	Specifies function inlining. See section 2.9.1, *Inlining Functions*.

`macro` `nomacro`	Turns macro expansion on (default) or off.
`maxcalldepth value`	With this pragma you can control the maximum call depth. Default is infinite (-1).
`message "string" ...`	Emits a message to standard output
`novector`	Do not generate channel vectors and channel context. See compiler option **--novector**.

Pragma name	Description
`protect` `endprotect`	Protect sections against linker optimizations. This excludes a section from unreferenced section removal and duplicate section removal by the linker.
`optimize` *`flags`* `endoptimize`	Controls compiler optimizations. See section 4.3, *Compiler Optimizations* in Chapter *Using the Compiler*
`section` `endsection`	Changes section names. See section 2.10, *Compiler Generated Sections* and compiler option **–R** in section 5.1, *Compiler Options* in Chapter *Tool Options* of the *Reference Manual*
`source` `nosource`	Specifies which C source lines must be shown in assembly output. See compiler option **–s** in section 5.1, *Compiler Options* in Chapter *Tool Options* of the *Reference Manual*.
`stdinc`	Changes the behaviour of the #include directive. When set, compiler option **–I** and compiler option **––no–stdinc** are ignored.

`binary_switch`
`jump_switch`
`linear_switch`
`smart_switch`
	Specifies switch statement. See section 2.11, *Switch Statement*

`tradeoff` *`level`*	Specify tradeoff between speed (0) and size (4). See compiler option **–t** in section 5.1, *Compiler Options* in Chapter *Tool Options* of the *Reference Manual*.
`warning [`*`number`*`,...]`	Disables warning messages. See compiler option **–w** in section 5.1, *Compiler Options* in Chapter *Tool Options* of the *Reference Manual*.
`weak` *`symbol`*	Marks a symbol as 'weak'

Table 2-3: Pragmas

2.7 PREDEFINED MACROS

In addition to the predefined macros required by the ISO C standard, the TASKING PCP C compiler supports the predefined macros as defined in Table 2-4. The macros are useful to create conditional C code.

Macro	Description
__BIGENDIAN__	Expands to 0.
__BUILD__	Expands to the build number of the compiler: BRRRrrr. The B is the build number, RRR is the major branch number and rrr is the minor branch number. Examples: Build #134 -> 134000000 Build #22.1.4 -> 22001004
__CPCP__	Identifies the compiler. You can use this symbol to flag parts of the source which must be recognized by the **cpcp** assembler only. It expands to 1.
__CPU__	Expands to a string with the CPU supplied with the option **--cpu**. When no **--cpu** is supplied, this symbol is not defined.
__CORE__	Expands to a string with the core depending on the C compiler options option **--cpu**. The symbol expands to "pcp2" when the option **--cpu** is not specified.
__DATE__	Expands to the compilation date: "mmm dd yyyy".
__DOUBLE_FP__	Expands to 0. The PCP always treats a `double` as `float`.
__FILE__	Expands to the current source file name.
__LINE__	Expands to the line number of the line where this macro is called.
__REVISION__	Expands to the revision number of the compiler. Digits are represented as they are; characters (for prototypes, alphas, betas) are represented by -1. Examples: v1.0r1 -> 1, v1.0rb -> -1
__SFRFILE__(*cpu*)	This macro expands to the filename of the used SFR file, including the < >. The *cpu* is the argument of the macro. For example, if **--cpu=tc1920b** is specified, the macro `__SFRFILE__(__CPU__)` expands to `__SFRFILE__(tc1920b)`, which expands to `<regtc1920b.sfr>`.
__SINGLE_FP__	Expands to 1. The PCP compiler always treats a `double` as `float`.

Macro	Description
__STDC__	Identifies the level of ANSI standard. The macro expands to 1 if you set option **--language** (Control language extensions), otherwise expands to 0.
__STDC_HOSTED__	Always expands to 0, indicating the implementation is not a hosted implementation.
__STDC_VERSION__	Identifies the ISO C version number. Expands to 199901L for ISO C99 or 199409L for ISO C90.
__TASKING__	Identifies the compiler as a TASKING compiler. Expands to 1 if a TASKING compiler is used.
__TASKING_SFR__	Expands to 1 if TASKING `.sfr` files are used. Not defined when option **--no-tasking-sfr** is used.
__TIME__	Expands to the compilation time: ”hh:mm:ss”
__VERSION__	Expands to a number to identify the compiler version: Mmmm The M is the major version number and the mmm is a three-digit minor version number. Examples: 1.0 -> 1000 v12.3 -> 12003

Table 2-4: Predefined macros

2.8 POINTERS

Objects receive a default addressing type, based on the memory model. No additional pointer qualifiers are added for the PCP to override the default addressing type.

The PCP compiler supports code and data pointers that need no further qualification.

Pointer	Location	Max. object size	Pointer size	Section name + type
data	PRAM	64 kB	14 bit	data
code	CMEM	128 kB	16 bit	code

The default section name is equal to the generated section type that is prefixed with `.pcptext.` for code and `.pcpdata.` for data. You can change section names with the #pragma section or with the command line option --rename-sections.

2.9 FUNCTIONS

2.9.1 INLINING FUNCTIONS: INLINE

With the compiler option **--optimize=+inline** (**-Oi**), the compiler automatically inlines small functions to reduce execution time. The compiler inserts the function body at the place the function is called. If the function is not called at all, the compiler does not generate code for it.

With the `inline` keyword you tell the compiler to inline the function body instead of calling the function. Use the `__noinline` keyword to tell the compiler *not* to inline the function body. These keywords overrule the compiler option **--optimize=+inline**.

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).

Example: inline

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

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

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

The function `add()` is defined before it is called. The compiler inserts (optimized) code for both calls to the `add()` function. The generated assembly is:

```
main:
    mov16  d15,#3
    st.w   w,d15

    ld.w   d15,x
    ld.w   d0,y
    add16  d0,d15
    st.w   z,d0
```

Example: #pragma inline / #pragma noinline

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

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

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

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

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

If you set `#pragma inline` at the beginning of a source file, *all* functions without the function qualifier `__noinline` are inlined. This is the same as using compiler option **--inline**.

#pragma smartinline

With the compiler option **--optimize=+inline** (**-Oi**), the compiler inlines small fuctions that are not too often called. This reduces execution time at the cost of code size.

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

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

See for more information of these options, section 5.1, *Compiler Options* in Chapter *Tool Options* of the *PCP Reference Manual*.

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 2.5, *Using Assembly in the C Source*, for more information about the `__asm` keyword.
Example 6 in that section shows how in combination with the `inline` keyword an intrinsic function is created.

2.9.2 INTERRUPT FUNCTIONS: __INTERRUPT()

With the function qualifier __interrupt you can declare a function as interrupt function (interrupt service routine). This function qualifier takes one argument:

```
__interrupt(CN)
```

The *CN* 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 set to zero, because a channel cannot be interrupted by another channel.
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, the Channel Enable bit is set (R7.CEN), because its value is not preserved during the execution of the channel, and an EXIT instruction is generated to stop channel execution. The arguments of the EXIT instruction generated are: EC=0, ST=1, INT=0, EP=1, cc_UC.

Example:

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

2.9.3 PARAMETER PASSING

The parameter registers R1, R3, R4, R6, R0 are used to pass the initial function arguments. The paramaters are passed from left to right.

The first unused and register that fits 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 address, each parameter on the stack is stored in little endian. The alignment on the stack depends on the data type as listed in table 2-1 in Section 2.2, Data Types.

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.

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 in direct function calls, due to the static stack implementation.

The following table summarize the registers used by the PCP compiler **cpcp**:

Register	Class	Usage
R0	caller saves	Parameter passing, 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, automatic variables
R4	caller saves	Parameter passing and automatic variables
R5	caller saves	Automatic variables and function return code compaction
R6	caller saves	Parameter passing and automatic variables and return buffer
R7	special purpose	PC, CC, DPTR

Table 2-5: Register usage

Registers are classified: *caller saves*, *callee saves*, and *special purpose*:

Class	Usage
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	Registers must be saved by the called function, i.e. the caller expects them not to bechanged after the function call.
special purpose	The purpose of R7 is defined by the PCP core.

Table 2-6: Register classes

Stack Usage

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

2.10 COMPILER GENERATED SECTIONS

Section names

The compiler uses the section type as default section names. The section names are prefixed with ELF pcp space names.

Section name syntax:

.pcp_space_name.section_type

The **pcp** space names are: `pcpdata` or `pcptext`.

The section types are: `code` or `data`.

The 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 to the function name in which the overlay section is allocated.

Overlay section name syntax:

.pcpdata.*function_name*

Renaming sections

You can rename sections with a pragma or on the command line.

Command line:

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

compiler option **--rename-sections** in section 5.1, *Compiler Options* in Chapter *Tool Options*.

Pragma:

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

With the section type you select which sections are renamed. The matching sections will get the specified format string for the section name. The format string 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
```

#pragma endsection

With the endsection pragma the default section name is restored:

Nesting of **#pragma section**/**#pragma endsection** pairs saves the status of the previous level.

Example (`example.c`):

```
char a;         // allocated in '.pcpdata.data'
#pragma section data=MyDataData1
char b;         // allocated in '.pcpdata.MyDataData1'
#pragma section data=MyDataData2
char c;         // allocated in '.pcpdata.MyDataData2'

#pragma endsection
char d;         // allocated in '.pcpdata.MyDataData1'
#pragma endsection
char e;         // allocated in '.pcpdata.data'
```

2.11 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.

`#pragma smart_switch` is the default of the compiler. The compiler tries to use the switch method which uses the least space in memory plus code to do the indexing.

Especially for large switch statements, the jump table approach executes faster than the binary search 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.

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

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

Using a pragma cannot overrule the restrictions as described earlier.

The switch pragmas must be placed before the function body containing 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 */
    }
}
```

2.12 LIBRARIES

The compiler **cpcp** comes with standard C libraries (ISO/IEC 9899:1999) and header files with the appropriate prototypes for the library functions. The standard C libraries are available in object format and in C or assembly source code.

A number of standard operations within C are too complex to generate inline code for. These operations are implemented as *run-time* library functions.

The directory structure is:

```
\pcp\lib\
        pcp1\      PCP1 libraries
        pcp15      PCP1.5 libraries
        pcp2\      PCP2 libraries
```

2.12.1 OVERVIEW OF LIBRARIES

Table 2-7 lists the libraries included in the PCP toolchain.

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

Table 2-7: Overview of libraries

See section 2.2, *Library Functions*, in Chapter *Libraries* of the *Reference Manual* for an extensive description of all standard C library functions.

2.12.2 PRINTF AND SCANF FORMATTING ROUTINES

The C library functions `printf()`, `fprintf()`, `vfprintf()`, `vsprintf()`, ... call one single function, `_doprint()`, that deals with the format string and arguments. The same applies to all scanf type functions, which call the function `_doscan()`, and also for the `wprintf` and `wscanf` type functions which call `_dowprint()` and `_dowscan()` respectively. The C library contains three versions of these routines: `int`, `long` and `long long` versions. If you use floating-point, the formatter function for floating-point `_doflt()` or `_dowflt()` is called. Depending on the formatting arguments you use, the correct routine is used from the library. Of course the larger the version of the routine the larger your produced code will be.

Note that when you call any of the printf/scanf routines indirect, the arguments are not known and always the `long long` version with floating-point support is used from the library.

Example:

```
#include <stdio.h>

long L;

void main(void)
{
    printf( "This is a long: %ld\n", L );
}
```

The linker extracts the `long` version without floating-point support from the library.

2.12.3 REBUILDING LIBRARIES

If you have manually changed one of the standard C library functions, you need to recompile the standard C libraries.

'Weak' symbols are used to extract the most optimal implementation of a function from the library. For example if your application does not use floating-point variables the `printf` alike functions do not support floating-point types either. The compiler emits strong symbols to guide this process. Do not change the order in which modules are placed in the library since this may break this process.

The sources of the libraries are present in the `lib\src` directory. This directory also contains subdirectories with a `makefile` for each type of library:

```
lib\src\
        pcp1\
            libc\makefile
            libfp\makefile
            libfpt\makefile
        pcp15\
            libc\makefile
            libfp\makefile
            libfpt\makefile
        pcp2\
            libc\makefile
            libfp\makefile
            libfpt\makefile
```

To rebuild the libraries, follow the steps below.

First make sure that the `bin` directory for the PCP toolchain is included in your PATH environment variable. (See section 1.3.1, *Configuring the Command Line Environment*.

1. Make the directory `lib\src\pcp1\libc` the current working directory.

   This directory contains a `makefile` *which also uses the default make rules from* `mkpcp.mk` *from the* `pcp\etc` *directory.*

2. Edit the `makefile`.

See section 7.3, *Make Utility*, in Chapter *Utilities* for an extensive description of the make utility and makefiles.

3. Assuming the `lib\src\pcp1\libc` directory is still the current working directory, type:

   `mkpcp`

   to build the library.

   The new library is created in the `lib\src\pcp1\libc` *directory.*

4. Make a backup copy of the original library and copy the new library to the `lib\pcp1` directory of the product.

CHAPTER 3

PCP ASSEMBLY LANGUAGE

TASKING

CHAPTER 3

3.1 INTRODUCTION

In this chapter the most important aspects of the PCP assembly language are described. For a complete overview of the PCP2 architecture, refer to the *PCP2 32-bit Single-Chip Microcontroller* [2000, Infineon].

3.2 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 and directives 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. 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. Note that if you use a reserved symbol as a label it has to be followed by a colon.

Examples:

```
      LAB1:  ; This label is followed by a colon and
               can start with a space or tab
   LAB1      ; This label has to start at the beginning
               of a line
```

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 3.4, *Operands of an Assembly Instruction*. The instructions are described in the *PCP Architecture Manuals*.

Examples:

```
jg      _PCP_multiply     ; One operand
ld.i    r3,0x5            ; Two operands
mov     r4,r6,cc_uc       ; Three operands
```

directive With directives you can control the assembler from within the assembly source. These must not start in the first column. Directives are described in section 3.8, *Assembler Directives and Controls*.

macro_call A call to a previously defined macro. It must not start in the first column. Macros are described in section 3.10 *Macro Operations*.

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 3.8, *Assembler Directives and Controls*.

3.3 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 3.6.3, *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

3.4 OPERANDS AND ADDRESSING MODES

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

Operand	Description
symbol	A symbolic name as described in section 3.5, *Symbol Names*. Symbols can also occur in expressions.
register	Any valid register or a register pair, register quad, register extension, register part or special function register.
expression	Any valid expression as described in the section 3.6, *Assembly Expressions*.
address	A combination of *expression*, *register* and *symbol*.

3.4.1 PCP 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 PCP/DMA Architecture manual from Siemens.

3.5 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.

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

Register names and names of assembler directives and controls are reserved for the system, so you cannot use these for user-defined symbols. The case of these built-in symbols is insignificant. Symbol names and other identifiers beginning with a period (.) are also reserved for the system.

The following symbols are predefined:

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.

Table 3-1: Predefined symbols

Examples

Valid symbol names	Invalid symbol names	
`loop_1`	`1_loop`	(starts with a number)
`ENTRY`	`r1`	(reserved register name)
`a_B_c`	`.space`	(reserved directive name)
`_aBC`		

3.6 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 are 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 contant*
- *string*
- *symbol*
- *expression binary_operator expression*
- *unary_operator expression*

- **(** *expression* **)**
- *function call*

All types of expressions are explained in separate sections.

3.6.1 NUMERIC CONSTANTS

Numeric constants can be used in expressions. If there is no prefix, the assembler assumes the number is a decimal number.

Base	Description	Example
Binary	'**0B**' or '**0b**' followed by binary digits (0,1).	`0B1101` `0b11001010`

Hexadecimal	'**0X**' or '**0x**' followed by a hexadecimal digits (0–9, A–F, a–f).	`0X12FF`
		`0x45`
		`0x9abc`

Decimal, integer	Decimal digits (0–9).	`12` `1245`

Decimal, floating–point	Includes a decimal point, or an '**E**' or '**e**' followed by the exponent.	`6E10`
		`.6`
		`3.14`
		`2.7e10`

3.6.2 STRINGS

ASCII characters, enclosed in single (') or double (") quotes constitue 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 the size of a long word (first 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 longer than 4 characters is used in a .BYTE assembler directive; in that case all characters result in a constant byte. 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           ; (0x00004143) 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'

['PCP assembler',0,3] ; results in the substring 'PCP'
```

3.6.3 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. Expressions between parentheses 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
	( )	parentheses	Expressions enclosed by parenthesis are evaluated first.
Unary	+	plus	Returns the value of its operand.
	–	minus	Returns the negative of its operand.
	~	complement	Returns complement, integer only
	!	logical negate	Returns 1 if the operands' value is 0; otherwise 0. For example, if `buf` is 0 then `!buf` is 1.
Arithmetic	*	multiplication	Yields the product of two operands.
	/	division	Yields the quotient of the division of the first operand by the second. With integers, the divide operation produces a truncated integer.
	%	modulo	Integer only: yields the remainder from a division of the first operand by the second.
	+	addtion	Yields the sum of its operands.
	–	subtraction	Yields the difference of its operands.
Shift	<<	shift left	Integer only: shifts the left operand to the left (zero-filled) by the number of bits specified by the right operand.
	>>	shift right	Integer only: shifts the left operand to the right (sign bit extended) by the number of bits specified by the right operand.
Relational	<	less than	If the indicated condition is:
	<=	less or equal	– True: result is an integer 1
	>	greater than	– False: result is an integer 0
	>=	greater or equal	Be cautious when you use floating-point values in an equality test; rounding errors can cause unexpected results.
	==	equal	
	!=	not equal	

Type	Operator	Name	Description
Bitwise	&	AND	Integer only: yields bitwise AND
	|	OR	Integer only: yields bitwise OR
	^	exclusive OR	Integer only: yields bitwise exclusive OR
Logical	&&	logical AND	Returns an integer 1 if both operands are nonzero; otherwise, it returns an integer 0.
	||	logical OR	Returns an integer 1 if either of the operands is nonzero; otherwise, it returns an integer 1

Table 3-2: Assembly expression operators

3.7 BUILT-IN ASSEMBLY FUNCTIONS

The assembler has several built-in functions to support data conversion, string comparison, and math computations. You can use functions as terms in any expression. Functions have the following syntax:

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

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

The following tables provide an overview of all built-in assembler functions. For a detailed description of these functions, see section 3.2, *Built-in Assembly Function*, in Chapter *Assembly Language* of the *Reference Manual*.

Overview of mathematical functions

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

Overview of conversion functions

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

Overview of string functions

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

Overview of macro functions

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

Overview of address calculation functions

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

Overview of assembler mode functions

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

3.8 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 directives

  - Data definition / Storage allocation directives
  - Debug directives
- Directives that are interpreted by the macro preprocessor. These directives tell the macro preprocessor how to manipulate your assembly code before it is actually being assembled. You can use these directives to write macros and to write conditional source code. Parts of the code that do not match the condition, will not be assembled at all.
- Some directives act as assembler options and most of them indeed do have an equivalent assembler (command line) option. The advantage of using a directive is that with such a directive you can overrule the assembler option for a particular part of the code. Directives of this kind are called *controls*. A typical example is to tell the assembler with an option to generate a list file while with the controls $LIST ON and $LIST OFF you overrule this option for a part of the code that you do *not* want to appear in the list file. Controls always appear on a separate line and start with a '$' sign in the first column.

  The following controls are available:

  - Assembly listing controls
  - Miscellaneous controls

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

3.8.1 OVERVIEW OF ASSEMBLER DIRECTIVES

The following tables provide an overview of all assembler directives. For a detailed description, see section 3.3.2, *Detailed Description of Assembler Directives*, in Chapter *Assembly Language* of the *Reference Manual*.

Overview of assembly control directives

Directive	Description
.COMMENT	Start comment lines. You cannot use this directive in .IF/.ELSE/.ENDIF constructs and .MACRO/.DUP definitions.
.DEFINE	Define substitution string
.END	End of source program
.FAIL	Programmer generated error message
.INCLUDE	Include file
.MESSAGE	Programmer generated message
.ORG	Initialize memory space and location counters to create a nameless section
.SDECL	Declare a section with name, type and attributes
.SECT	Activate a declared section
.UNDEF	Undefine .DEFINE symbol
.WARNING	Programmer generated warning

Overview of symbol definition directives

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

Overview of data definition / storage allocation directives

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

Overview of macro and conditional assembly directives

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

Overview of debug directives

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

3.8.2 OVERVIEW OF ASSEMBLER CONTROLS

The following tables provide an overview of all assembler controls. For a detailed description, see section 3.3.4, *Detailed Description of Assembler Controls*, in Chapter *Assembly Language* of the *Reference Manual*.

Overview of assembler listing controls

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

Overview of miscellaneous assembler controls

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

3.9 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 6.7.6 *The Section Layout Definition: Locating Sections* in chapter *Using the Linker*.

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 .SDECL directive in section 3.3.2, *Detailed Description of Assembler Directives*, in chapter *Assembly Language* of the *Reference Manual*, 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. By using this naming scheme it is possible to collect all pieces of code or data belonging together into one bigger section during the linking phase. A .SECT directive referring to an earlier defined section is called a *continuation*. Only the name can be specified.

Example 1

```
.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 with the locator description file.

Example 2

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

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

3.10 MACRO OPERATIONS

Macros provide a shorthand method for inserting a repeated pattern of code or group of instructions. Yuo 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 one 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.

3.10.1 DEFINING A MACRO

The first step in using a macro is to define it in the source file. The definition of a macro consists of three parts:

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

A macro definition takes the following form:

```
Header:     macro_name .MACRO [arg[,arg...] [; comment]
                       .
Body:                  source statements
                       .
Terminator:            .ENDM
```

If the macro name is the same as an existing assembler directive or mnemonic opcode, the assembler replaces the directive or mnemonic opcode with the macro and issues a warning.

The arguments are symbolic names that the macro preprocessor replaces with the literal arguments when the macro is expanded (called). Each argument must follow the same rules as global symbol names. Argument names cannot start with a percent sign (**%**).

Example

The macro definition:

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

The macro call:

```
    .SDECL '.pcptext', code
    .SECT '.pcptext'

    CONSTD  r5,0x12345678

    .END
```

The macro expands as follows:

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

3.10.2 CALLING A MACRO

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

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

where:

label	An optional label that corresponds to the value of the location counter at the start of the macro expansion.
macro_name	The name of the macro. This must be in the operation field.
arg	One or more optional, substitutable arguments. Multiple arguments must be separated by commas.
comment	An optional comment.

The following applies to macro arguments:

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

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

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

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

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

3.10.3 USING OPERATORS FOR MACRO ARGUMENTS

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

Operator	Name	Description
\	Macro argument concatenation	Concatenates a macro argument with adjacent alphanumeric characters.
?	Return decimal value of symbol	Substitutes the **?***symbol* sequence with a character string that represents the decimal value of the symbol.
%	Return hex value of symbol	Substitutes the **%***symbol* sequence with a character string that represents the hexadecimal value of the symbol.
"	Macro string delimiter	Allows the use of macro arguments as literal strings.
^	Macro local label override	Causes local labels in its term to be evaluated at normal scope rather than at macro scope.

*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.W  D0,[RREG1]
     LD.W  D1,[RREG2]
     ST.W  [RREG1],D1
     ST.W  [RREG2],D0
```

which results in an assembler error.

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.W  D0,_lab\?REG1            ;use D0 as temp
     LD.W  D1,_lab\?REG2            ;use D1 as temp
     ST.W  _lab\?REG1,D1
     ST.W  _lab\?REG2,D0
     .ENDM

     LD.P    R4,[R\?REG1],CC_UC
     LD.P    R5,[R\?REG2],CC_UC
     ST.P    R5,[R\?REG1],CC_UC
     ST.P    R4,[R\?REG2],CC_UC
```

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
```

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
```

A symbol with the name NUM is set to 10 and the macro is called with NUM as argument:

```
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`.

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 quotes and double quotes to avoid inappropriate expansions. Since a `.DEFINE` expansion occurs before a macro substitution, all `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
```

The macro expands as:

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

Single quotes prevent expansion so the first **.MESSAGE** is not stated as is. In the double quoted **.MESSAGE**, first the define LONG is expanded to 'short' and then the argument STRING is substituted by '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_L0000001**).

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
       JC ^LAB,CC_NZ
       .ENDM
```

The macro is called as follows:

```
      INIT 2,4
```

The macro expands as:

```
       LD.I    R5,0X1
LAB:
       .WORD 2
       ADD.I   R5,0X1
       COMP.I  R5,#4
       JC      LAB,CC_NZ
```

Without the ^ operator, the macro preprocessor would choose another name for LAB because the label already exists. The macro then 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
```

3.10.4 USING THE .DUP, .DUPA, .DUPC, .DUPF DIRECTIVES AS MACROS

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.

For a detailed description of these directives, see section 3.3, *Assembler Directives*, in Chapter *Assembly Language* of the *Reference Manual*.

3.10.5 CONDITIONAL ASSEMBLY: .IF, .ELIF AND .ELSE DIRECTIVES

With the conditional assembly directives you can instruct the macro preprocessor to use a part of the code that matches a certain condition.

You can specify assembly conditions with arguments in the case of macros, or through definition of symbols via the `.DEFINE`, `.SET`, and `.EQU` directives.

The built-in functions of the assembler provide a versatile means of testing many conditions of the assembly environment.

You can use conditional directives also within a macro definition to check at expansion time if arguments fall within a certain range of values. In this way macros become self-checking and can generate error messages to any desired level of detail.

The conditional assembly directive **.IF** has the following form:

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

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.

For a detailed description of these directives, see section 3.3, *Assembler Directives*, in Chapter *Assembly Language* of the *Reference Manual*.

CHAPTER 4

USING THE COMPILER

TASKING

4 CHAPTER

4.1 INTRODUCTION

On the command line it is possible to call the compiler separately from the other tools. However, it is recommended to use the control program **ccpcp** for command line invocations of the toolchain (see section 7.2, *Control Program*, in Chapter *Using the Utilities*). With the control program it is possible to call the entire toolchain with only one command line.

The compiler takes the following files for input and output:

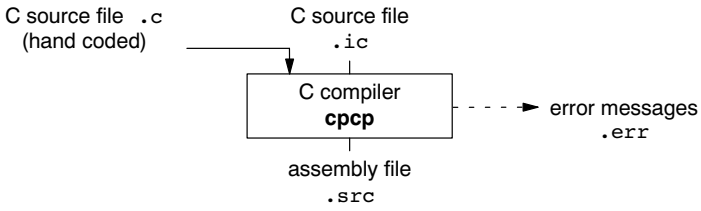

Figure 4-1: C compiler

This chapter first describes the compilation process which consists of a *frontend* and a *backend* part. During compilation the code is optimized in several ways. The various optimizations are described in the second section. Third it is described how to call the compiler and how to use its options. An extensive list of all options and their descriptions is included in the section 5.1, *Compiler Options*, in Chapter 5, *Tool Options*, of the *Reference Manual*. Finally, a few important basic tasks are described.

4.2 COMPILATION PROCESS

During the compilation of a C program, the compiler **cpcp** runs through a number of phases that are divided into two groups: *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 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:

   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

5. Instruction selector phase:

   This phase reads the MIL input and translates it into Low level Intermediate Language (LIL). The LIL objects correspond to a PCP processor instruction, with an opcode, operands and information used within the compiler.

6. 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.

7. Register allocator phase:

   This phase chooses a physical register to use for each virtual register.

8. The backend optimization phase:

   Performs target processor *independent* and *dependent* optimizations which operate on the Low level Intermediate Language.

9. The code generation/formatter phase:

   This phase reads through the LIL operations to generate assembly language output.

4.3 COMPILER OPTIMIZATIONS

The compiler has a number of optimizations which you can enable or disable. To enable or disable optimizations:

```
cpcp test.c -Oflag
```

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 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 below. The command line option for each optimization is given in brackets.

See also option **-O** (**--optimize**) in section 5.1, *Compiler Options*, of Chapter *Tool Options* of the *PCP Reference Manual*.

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).

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.

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.

Automatic Memory Partition *(option* ***–Ob****/****–OH****)*

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.

Loop transformations *(option* ***–Ol****/****–OL****)*

Temporarily 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.

Constant propagation *(option* ***–Op****/****–OP****)*

A variable with a known constant value is replaced by that value.

Subscript strength reduction *(option* ***–Os****/****–OS****)*

An array of 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.

Core specific optimizations (backend)

Coalesce *(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 and 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.

Generic assembly optimization *(option **-Og**/**-OG**)*

A set of target independent optimizations that increase speed and decrease code size.

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.

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.

4.3.1 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 focuses on code size optimization.

To specify the size/speed trade-off optimization level:

```
cpcp test.c -tlevel
```

See also option **-t** (**--tradeoff**) in section 5.1, *Compiler Options*, in Chapter *Tool Options* of the *PCP Reference Manual*.

4.4 CALLING THE COMPILER

The invocation syntax is:

```
cpcp [option]... [file]
```

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

```
cpcp test.c
```

This compiles the file `test.c` and generates the file `test.src` which serves as input for the assembler.

For a complete overview of all options with extensive description, see section 5.1, *Compiler Options*, of Chapter *Tool Options* of the *PCP Reference Manual*.

4.5 HOW THE COMPILER SEARCHES INCLUDE FILES

When you use include files, you can specify their location in several ways. The compiler searches the specified locations in the following order:

1. If the #include statement contains a pathname, the compiler looks for this file. If no 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 **Directories** dialog (**-I** 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 which paths were set during installation. You can still change these paths.

See environment variable CPCPINC in section 1.3.1, *Configuring the Command Line Environment*, in Chapter *Software Installation*.

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 **`--nostdinc`**).

4.6 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.

```
cpcp -g
```

Due to different compiler opimizations, it might be possible that certain debug information is optimized away. Therefore, it is best to specify **No optimization** (**-O0**) when you want to debug your application.

4.7 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 9, *MISRA-C Rules*, in the *Reference Manual*.

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 ggenerated 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 PCP 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.

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

See compiler option **--misrac** in section 5.1, *Compiler Options* in Chapter *Tool Options* of the *PCP Reference Manual*.

See linker option **--misra-c-report** in section 5.3, *Linker Options* in Chapter *Tool Options* of the *PCP Reference Manual*.

4.8 C COMPILER ERROR MESSAGES

The **cpcp** compiler reports the following types of error messages:

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 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 with compiler option **-w**.

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.

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

See compiler option **--diag** in section 5.1, *Compiler Options* in Chapter *Tool Options* of the *PCP Reference Manual*.

CHAPTER 5

USING THE ASSEMBLER

TASKING

CHAPTER 5

5.1 INTRODUCTION

The assembler converts hand-written or compiler-generated assembly language programs into machine language, using the Executable and Linking Format (ELF) for object files.

The assembler takes the following files for input and output:

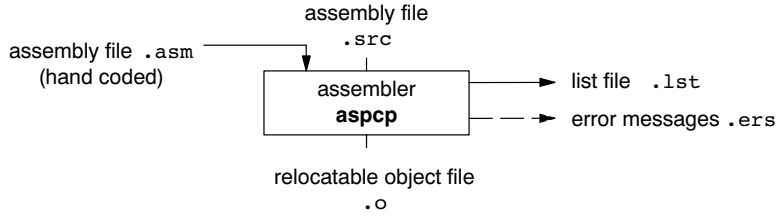

Figure 5-1: Assembler

This chapter first describes the assembly process. The various assembler optimizations are described in the second section. Third it is described how to call the assembler and how to use its options. An extensive list of all options and their descriptions is included in the *Reference Manual*. Finally, a few important basic tasks are described.

5.2 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

1. Preprocess directives
2. Check syntax of instructions
3. Instruction grouping and reordering
4. Optimization (instruction size and generic instructions)
5. Generation of the relocatable object file and optionally a list file

   The assembler integrates file inclusion and macro facilities. See section 3.10, *Macro Operations*, in Chapter *PCP Assembly Language* for more information.

5.3 ASSEMBLER OPTIMIZATIONS

The **aspcp** assembler performs various optimizations to reduce the size of the assembled applications. There are two options available to influence the degree of optimization.

See also option **-O** (**--optimize**) in section 5.2, *Assembler Options*, in Chapter *Tool Options* of the *PCP Reference Manual*.

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 the generic instructions by faster or smaller instructions.
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.

5.4 CALLING THE ASSEMBLER

The invocation syntax is:

```
aspcp [option]... [file]
```

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

```
aspcp test.asm
```

This assembles the file `test.asm` and generates the file `test.o` which serves as input for the linker.

For a complete overview of all options with extensive description, see section 5.2, *Assembler Options*, of Chapter *Tool Options* of the *PCP Reference Manual*.

5.5 SPECIFYING A TARGET PROCESSOR

Before you call the assembler, you need to tell the assembler for which target processor it needs to assemble. Of course you need to specify a processor with a PCP such as the `tc1920b`. Based on the processor type, the assembler includes a *special function register file*. This is a regular include file which enables you to use virtual registers that are located in memory.

```
aspcp -Ctc1920b test.src
```

5.6 HOW THE ASSEMBLER SEARCHES INCLUDE FILES

When you use include files, you can specify their location in several ways. The assembler searches the specified locations in the following order:

1. If the `.INCLUDE` directive contains a pathname, the assembler looks for this file. If no 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 **Directories** dialog (**–I** 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 which paths were set during installation. You can still change these paths if you like.

See environment variable ASPCPINC in section 1.3.1, *Configuring the Command Line Environment*, in Chapter *Software Installation*.

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

5.7 GENERATING A LIST FILE

The list file is an additional output file that contains information about the generated code. With the options in the **List File** page of the **Assembler** entry in the **Project Options** dialog you choose to generate a list file or to skip it (**-l** option). You can also customize the amount and form of information (**-L** option).

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.

See section 6.1, *Assembler List File Format*, in Chapter *List File Formats* of the *Reference Manual* for an explanation of the format of the list file.

Example:

```
aspcp -l test.src
```

With this command the list file `test.lst` is created.

5.8 ASSEMBLER ERROR MESSAGES

The assembler produces error messages of the following types:

F Fatal errors

After a fatal error the assembler immediately aborts the assembling 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 with assembler option **-w**.

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

See assembler option **--diag** in section 5.2, *Assembler Options* in Chapter *Tool Options* of the *PCP Reference Manual*.

CHAPTER 6

USING THE LINKER

TASKING

9 CHAPTER

6.1 INTRODUCTION

The linker **lpcp** 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 takes the following files for input and output:

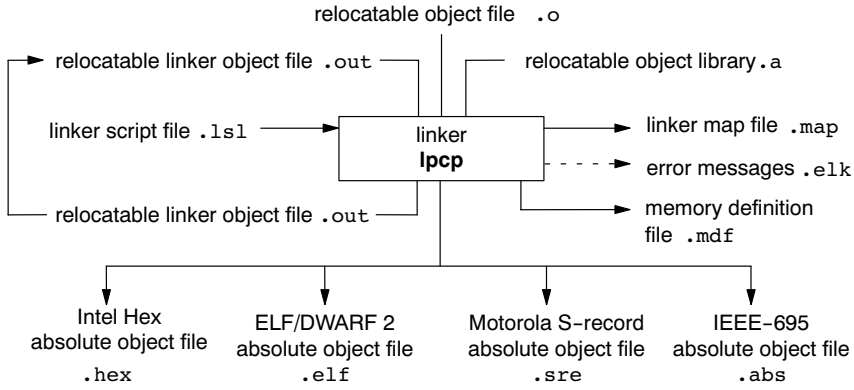

Figure 6-1: lpcp Linker

This chapter first describes the linking process. Then it describes how to call the linker and how to use its options. An extensive list of all options and their descriptions is included in section 5.3, *Linker Options*, of the *Reference Manual*.

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 Chapter 8, *Linker Script Language*, of the *Reference Manual*.

The end of the chapter describes how to generate a map file and contains an overview of the different types of messages of the linker.

6.2 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.

Glossary of terms

Term	Definition
Absolute object file	Object code in which addresses have fixed absolute values, ready to load into a target.
Address	A specification of a location in an address space.
Address space	The set of possible addresses. A core can support multiple spaces, for example in a Harvard architecture the addresses that identify the location of an instruction refer to code space, whereas addresses that identify the location of a data object refer to a data space.
Architecture	A description of the characteristics of a core that are of interest for the linker. This encompasses the address space(s) and the internal bus structure. Given this information the linker can convert logical addresses into physical addresses.
Copy table	A section created by the linker. This section contains data that specifies how the startup code initializes the data and BSS sections. For each section the copy table contains the following fields: – 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).

Term	Definition
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 addresses 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.
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 that may be shared between processors.
Unresolved reference	A reference to a symbol for which the linker did not find a definition yet.

Table 6-1: Glossary of terms

6.2.1 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 or data 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. 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.

6.2.2 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/DWARF2 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 **-o** (**--output**) and **-c** (**--chip-output**).

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 **lpcp** 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 6.7, *Controlling the Linker with a Script*.

6.2.3 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 use the linker option **–O** (**––optimize**) in section 5.3, *Linker Options*, in Chapter *Tool Options* of the *PCP Reference Manual*.

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 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.

Copy table compression *(option **–Ot**/**–OT**)*

The startup code initializes the application's data and BSS areas. The information about which memory addresses should be zeroed (bss) 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.

This optimization reduces both memory and startup time.

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

This optimization removes unused sections from the resulting object file. Because debug information normally refers to all sections, this optimization has no effect until you compile your project without debug information or use linker option **--strip-debug** to remove the debug information.

Delete duplicate code sections *(option **-Ox**/**-OX**)*

Delete duplicate data sections *(option **-Oy**/**-OY**)*

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

6.3 CALLING THE LINKER

The invocation syntax on the command line is:

```
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 6.4.1, *Specifying Libraries to the Linker*

For a complete overview of all options with extensive description, see section 5.3, *Linker Options*, of the *Reference Manual*.

6.4 LINKING WITH LIBRARIES

There are two kinds of libraries: system libraries and user libraries.

System library

The system libraries are installed in subdirectories of the `lib` directory of the toolchain. An overview of the system libraries is given in the following table.

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

Table 6-2: Overview of libraries

For more information on these libraries see section 2.12, *Libraries*, in Chapter *PCP C Language*.

When you want to link system libraries, you must specify this with the option **-l**. With the option **-lc** you specify the system library `libc.a`.

User library

You can also create your own libraries. Section 7.4, *Archiver*, in Chapter *Using the Utilities*, describes how you can use the archiver to create your own library with object modules. To link user libraries, specify their filenames on the command line.

6.4.1 SPECIFYING LIBRARIES TO THE LINKER

When you want to link system libraries, you must specify them with the linker option **-l**. With the option **-lc** you specify the system library `libc.a`. For example:

```
lpcp -lc start.o
```

To link the user library `mylib.a`:

```
lpcp start.o mylib.a
```

If the library resides in a subdirectory, specify that directory with the library name:

```
lpcp start.o mylibs\mylib.a
```

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 on 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.

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`.

6.4.2 HOW THE LINKER SEARCHES LIBRARIES

System libraries

You can specify the location of system libraries (specified with option **-l**) in several ways. The linker searches the specified locations in the following order:

1. The linker first looks in the directories that are specified with the **-L** 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 which paths were set during installation. You can still change these paths if you like.

3. When the linker did not find the library, it tries the default `lib` directory which was created during installation (or a processor specific sub-directory).

User library

If you use your own library, the linker searches the library in the current directory only.

6.4.3 HOW THE LINKER EXTRACTS OBJECTS FROM LIBRARIES

A library built with **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 **-v** option 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 symbol definition(s) 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 **-e**:

```
lpcp -e 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.

6.5 INCREMENTAL LINKING

With the PCP linker **lpcp** 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 and the `.out` file you had created with the first invocation.

Incremental linking is only possible on the command line.

```
lpcp -r 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 `task1.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 **-r** for incremental linking also suppresses warnings and errors because of unresolved symbols.

6.6 LINKING THE C STARTUP CODE

You need the run-time startup code to build an executable application. The default startup code acts as a 'wrapper' which places the main() routine of the PCP application in an interrupt service routine on channel 1 which is invoked from the TriCore application whenever the interrupt is triggered.

The startup code is part of the C library `libc.a`, and the source is present in the file `cstart.c` in the directory `lib\src`.

See section 4.2, *Startup Code*, in Chapter *Run-time Environment* of the *Reference Manual* for a description of the C startup code.

6.7 CONTROLLING THE LINKER WITH A SCRIPT

With the options on the command line you can control the linker's behavior to a certain degree. 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 toolchain.

6.7.1 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 and its *internal* memory (this is called the derivative). These definitions are written by Altium and supplied with the toolchain.

2. It provides the linker with a specification of the *external* memory attached to the target processor. The template `extmem.lsl` is supplied with the toolchain.

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 PCP architectures and derivatives that Altium has supplied in the `include.lsl` directory. Do not change these files.

If you attached external memory to a derivative you must specify this in a separate LSL file and pass both the LSL file that describes the derivative's architecture and your LSL file that contains the memory specification to the linker. Next you may also 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 syntax is described in Chapter 8, *Linker Script Language*, in the *Reference Manual*.

6.7.2 STRUCTURE OF A LINKER SCRIPT FILE

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

The architecture definition (required)

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

This specification is normally written by Altium. For each TriCore core architecture, a separate LSL file is provided. These are `tc1v1_2.lsl`, `tc1v1_3.lsl`, and `tc2.lsl`. These files include and extend the generic architecture file `tc_arch.lsl`. The PCP architecture is part of the TriCore `.lsl` files.

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 derivative definition (required)

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

Altium provides LSL descriptions of supported derivatives, along with "SFR files", which provide easy access to registers in I/O sub-systems from C and assembly programs. When you build an ASIC or use a derivative that is not (yet) supported by the TASKING tools, you may have to write a derivative definition.

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

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, for example with **-dtc1920b.lsl**.

The memory and bus definitions (optional)

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

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 a physical addresses (offsets 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.

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 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'
}
```

6.7.3 THE ARCHITECTURE DEFINITION: SELF-DESIGNED CORES

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
- 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. An address space is a range of addresses starting from zero. Normally, the size of an address space is to 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 TriCore as defined in the LSL file `tc_arch.lsl`.

Space	Id	MAU	ELF sections
linear	1	8	.text, .bss, .data, .rodata, table, istack, ustack
abs24	2	8	
abs18	3	8	.zdata, .zbss
csa	4	8	csa.* (Context Save Area)
pcp_code	8	16	.pcptext
pcp_data	9	32	.pcpdata

Table 6-3: TriCore address spaces

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:

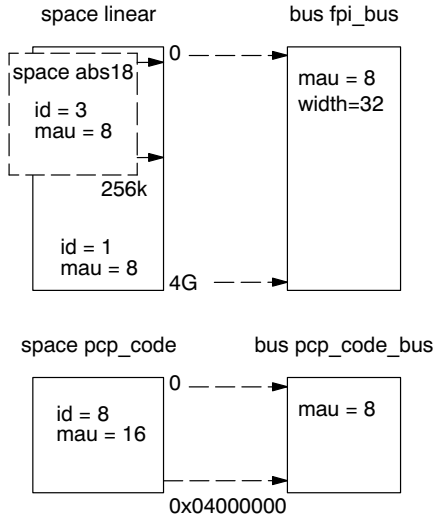

Figure 6-2: Scheme of (part of) the TriCore architecture

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);
}
```

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.
}
```

6.7.4 THE DERIVATIVE DEFINITION: SELF-DESIGNED PROCESSORS

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

Core

Each derivative 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 must 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

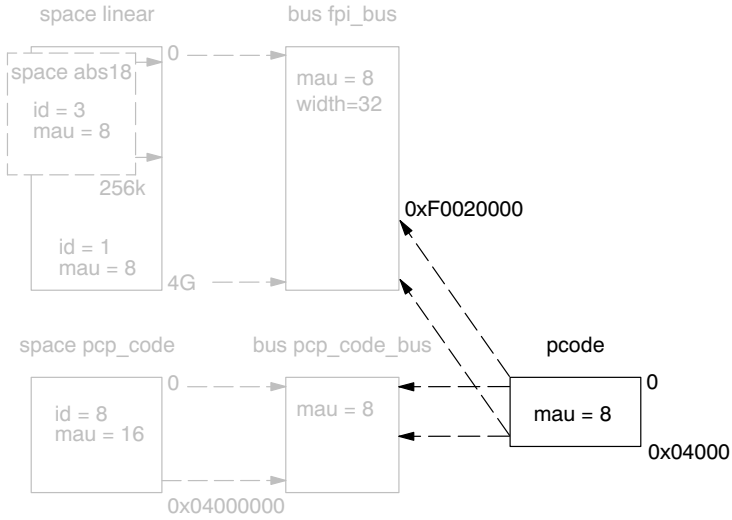

Figure 6-3: Internal memory definition for a derivative

According to the drawing, the TriCore contains internal memory called **pcode** with a size **0x04000** (16k). 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.
}
```

6.7.5 THE MEMORY DEFINITION: DEFINING EXTERNAL MEMORY

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
{
    External memory definitions.
}
```

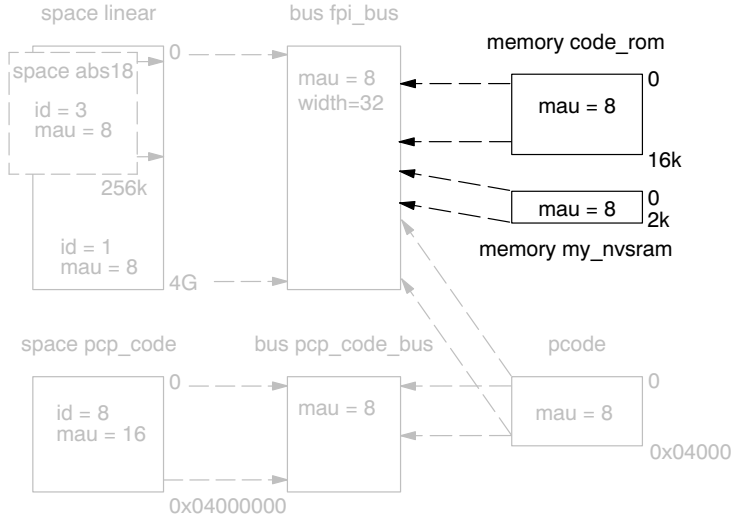

Figure 6-4: Adding external memory to the TriCore architecture

Suppose your embedded system has 16k of external ROM, named **code_rom** and 2k of external NVRAM, named **my_nvsram**. (See figure above.) Both memories are connected to the bus **fpi_bus**. In LSL this looks like follows:

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

The memory **my_nvsram** is connected to the bus with an offset of 0xc0000000:

```
memory my_nvsram
{
        mau = 8;
        size = 2k;
        type = ram;
        map (dest=bus:X: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 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.

6.7.6 THE SECTION LAYOUT DEFINITION: LOCATING SECTIONS

Once you have defined the internal core architecture and optional external 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 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. To illustrate section placement the following example of a C program is used:

Example: section propagation through the toolchain

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 section all

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 date="name"` the compiler's default section naming convention is overruled and a section with the name `non_volatile` is defined. In this section the battery back-upped data is stored.

By default the compiler creates the section `'.pcpdata.data'` to store uninitialized data objects.

As a result of the `#pragma section data="non_volatile"`, the data objects between the pragma pair are placed in `'.pcpdata.non_volatile'`. Note that sections are cleared at startup. However, battery back-upped sections should not be cleared and therefore we used the `#pragma noclear`.

The generated assembly may look like:

```
        .name   'test3.c'
        .sdecl  '_PCP_main', data, overlay('stack_data')
        .sect   '_PCP_main'
_PCP__999001__3:        .type   object
        .space  1
        ; End of section

        .sdecl  '.pcptext.data', code
        .sect   '.pcptext.data'
        .global _PCP_main
; Function _PCP_main
_PCP_main:      .type   func
        ldl.il  r7,@DPTR(_PCP__999001__3)
        st.pi   r2,[_PCP__999001__3]
        ldl.iu  r5,@HI(0xa5f0)
        ldl.il  r5,@LO(0xa5f0)
        ldl.il  r7,@DPTR(_PCP_battery_backup_tag)
        .
        .
        .
        jg      _PCP_printf
        ; End of function
        ; End of section

        .sdecl  '.pcpdata.data', data, clear
        .sect   '.pcpdata.data'
        .global _PCP_uninitialized_data
_PCP_uninitialized_data:        .type   object
        .size   _PCP_uninitialized_data,1
        .space  1
        ; End of section

        .sdecl  '.pcpdata.data', data
        .sect   '.pcpdata.data'
        .global _PCP_initialized_data
_PCP_initialized_data:  .type   object
        .size   _PCP_initialized_data,1
        .word   1
        ; End of section

        .sdecl  '.pcpdata.non_volatile', data, noclear,
group('page0_test3')
        .sect   '.pcpdata.non_volatile'
        .global _PCP_battery_backup_tag

_PCP_battery_backup_tag:        .type   object
        .size   _PCP_battery_backup_tag,1
        .space  1
        ; End of section
```

```
        .sdecl  '.pcpdata.non_volatile', data, noclear,
group('page0_test3')
        .sect   '.pcpdata.non_volatile'
        .global _PCP_battery_backup_invok
_PCP_battery_backup_invok:      .type   object
        .size   _PCP_battery_backup_invok,1
        .space  1
        ; End of section

        .sdecl  '.pcpdata.data', data
        .sect   '.pcpdata.data'
_PCP__1_str:    .type   object
        .size   _PCP__1_str,44
        .word   84   /* T */
        .word   104  /* h */
        ; This application has been invoked %d times\n
        .word   10   /* \n */
        .word   0    /* NULL */
        ; End of section

        .extern _PCP__cstart
        .calls  '_PCP_main','_PCP_printf'
        .extern _PCP_printf
        .sdecl  '_PCP_printf', data, max, overlay('stack_data')
        .sect   '_PCP_printf'
        .space  1
        ; End of section

        .extern _PCP___printf_int
        .extern _PCP__data__printf
        ; Module end
        .end
```

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_nvsram` from the example in the previous section.

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 `abs18`:

```
section_layout ::abs18
{
    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_nvsram`:

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

For a complete description of the Linker Script Language, refer to Chapter 8, *Linker Script Language*, in the *Reference Manual*.

6.7.7 THE PROCESSOR DEFINITION: USING MULTI-PROCESSOR SYSTEMS

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 proc_name
{
   derivative = deriv_name
}
```

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

6.8 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
`_lc_ub_`*name* `_lc_b_`*name*	Begin of section *name*. Also used to mark the begin of the stack or heap or copy table.
`_lc_ue_`*name* `_lc_e_`*name*	End of section *name*. Also used to mark the end of the stack or heap.
`_lc_cb_`*name*	Start address of an overlay section in ROM.
`_lc_ce_`*name*	End address of an overlay section in ROM.
`_lc_gb_`*name*	Begin of group *name*. This label appears in the output file even if no reference to the label exists in the input file.
`_lc_ge_`*name*	End of group *name*. This label appears in the output file even if no reference to the label exists in the input file.
`_lc_s_`*name*	Variable *name* is mapped through memory in shared memory situations.

Table 6-4: Linker labels

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.

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
`_lc_cp`	Start of copy table. Same as `_lc_ub_table`. The copy table gives the source and destination addresses of sections to be copied. This table will be generated by the linker only if this label is used.
`_PCP__lc_ub_heap`	Begin of heap. Same as `_lc_ub_pcp_heap`.
`_PCP__lc_ue_heap`	End of heap. Same as `_lc_ue_pcp_heap`.

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 */
}
#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:

```
#include <stdio.h>
extern void *_lc_ub_pcptext_myname;

int main(void)
{
    printf("The function myfunc is located at %X\n",
            &_lc_ub_pcptext_myname);
}
```

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 is defined and you can access it in the usual way. For the PCP, when you use the variable directly in your TriCore source, this would refer to 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 defind 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;

int main(int argc, char **argv)
{
    _lc_s_foo = 7;
}
```

Example: refer to the 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;
extern char *_lc_ue_heap;
int main()
{
  printf( "Size of the heap is %d\n",
          _lc_ue_heap - _lc_ub_heap );
}
```

From assembly you can refer to the end of the heap with:

```
.extern _lc_ue_heap   ; heap end
```

6.9 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

```
lpcp -Mtest.map test.o
```

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

See section 6.2, *Linker Map File Format*, in Chapter *List File Formats* of the *Reference Manual* for an explanation of the format of the map file.

6.10 LINKER ERROR MESSAGES

The linker produces error messages of the following types:

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 with linker option **-w**.

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 **-v**.

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.

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

See linker option **--diag** in section 5.3, *Linker Options* in Chapter *Tool Options* of the *PCP Reference Manual*.

CHAPTER 7

USING THE UTILITIES

TASKING

1 CHAPTER

7.1 INTRODUCTION

The TASKING toolchain for the PCP comes with a number of utilities that provide useful extra features.

ccpcp A control program for the PCP toolchain. The control program invokes all tools in the toolchain and lets you quickly generate an absolute object file from C source input files.

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 rebuild.

arpcp An ELF archiver. With this utility you create and maintain object library files.

7.2 CONTROL PROGRAM

The control program **ccpcp** is a tool that invokes all tools in the toolchain 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.

7.2.1 CALLING THE CONTROL PROGRAM

You can only call the control program from the command line. The invocation syntax is

```
ccpcp [ [option]... [file]... ]...
```

For example:

```
ccpcp -v test.c
```

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

```
+ c:\cpcp\bin\cpcp -o test.src test.c
+ c:\cpcp\bin\aspcp -o test.o test.src
+ c:\cpcp\bin\lpcp -o test.elf -ddefault.lsl
  -dextmem.lsl --map-file test.o -Lc:\cpcp\lib\pcp1
  -lc -lfp -lrt
```

By default, the control program removes the intermediate output files (`test.src` and `test.o` in the example above) afterwards, unless you specify the command line option **-t** (**--keep-temporary-files**).

For a complete list and description of all control program options, see section 5.4, *Control Program Options*, in Chapter *Tool Options* of the *Reference Manual*.

Recognized input files

The control program recognizes the following input files:

- Files with a `.c` suffix are interpreted as C source programs and are passed to the compiler.
- Files with a `.asm` suffix are interpreted as hand-written assembly source files which have to be passed to the assembler.
- Files with a `.src` suffix are interpreted as compiled assembly source files. They are directly passed to the assembler.
- Files with a `.a` suffix are interpreted as library files and are passed to the linker.
- Files with a `.o` suffix are interpreted as object files and are passed to the linker.
- Files with a `.out` suffix are interpreted as linked object files and are passed to the locating phase of the linker. The linker accepts only one `.out` file in the invocation.
- An argument with a `.lsl` suffix is interpreted as a linker script file and is passed to the linker.

The options in table NO TAG are options that the control program interprets itself. The control program however can also pass an option directly to a tool. Such an option is not interpreted by the control program but by the tool itself. The next example illustrates how an option is passed directly to the linker to link a user defined library:

```
ccpcp -Wl-lmylib test.c
```

Use the following options to pass arguments to the various tools:

Description	Option
Pass argument directly to the C compiler	**-Wc***arg*
Pass argument directly to the assembler	**-Wa***arg*
Pass argument directly to the PCP assembler	**-Wpcp***arg*
Pass argument directly to the linker	**-Wl***arg*

Table 7-1: Control program options to pass an option directly to a tool

If you specify an unknown option to the control program, the control program looks if it is an option for a specific tool. If so, it passes the option directly to the tool. However, it is recommended to use the control program options to passing arguments directly to tools.

With the environment variable CCPCPOPT you can define options and/or arguments that the control programs always processes *before* the command line arguments.

For example, if you use the control program always with the option **--no-map-file** (do not generate a linker map file), you can specify "--no-map-file" to the environment variable CCPCPOPT.

See section 1.3.1, *Configuring the Command Line Environment*, in Chapter *Software Installation*.

7.3 MAKE UTILITY

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 **ccpcp** 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, linked and located 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 7.3.2, *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 `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 -otest.elf test.o -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 toolchain.

Example

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

```
mkpcp test.elf

mkpcp -f mymake.mak test.elf
```

By default, the make utility reads `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** *my_makefile*.

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`.

The make utility now tries to build `test.elf` based on the `makefile` and peforms 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 `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 -otest.elf test.o -lc -lfp`.

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).

7.3.1 CALLING THE MAKE UTILITY

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

```
mkpcp [ [options] [targets] [macro=def]... ]
```

For example:

```
mkpcp 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 **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.

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.

Options of the make utility

The following make utility options are available:

Description	Option
Display options	**–?**
Display version header	**–V**
Verbose	
Print makefile lines while being read	**–D**/**–DD**
Display time comparisons which indicate a target is out of date	**–d**/**–dd**
Display current date and time	**–time**
Verbose option: show commands without executing (dry run)	**–n**
Do not show commands before execution	**–s**
Do not build, only indicate whether target is up–to–date	**–q**

Description	Option
Input files	
Use *makefile* instead of the standard makefile `makefile`	**-f** *makefile*
Change to directory before reading the makefile	**-G** *path*
Read options from file	**-m** *file*
Do not read the `mkpcp.mk` file	**-r**
Process	
Always rebuild target without checking whether it is out-of-date	**-a**
Run as a child process	**-c**
Environment variables override macro definitions	**-e**
Do not remove temporary files	**-K**
On error, only stop rebuilding current target	**-k**
Overrule the option **-k** (only stop rebuilding current target)	**-S**
Make all target files precious	**-p**
Touch the target files instead of rebuilding them	**-t**
Treat target as if it has just been reconstructed	**-W** *target*
Error messages	
Redirect error messages and verbose messages to a file	**-err** *file*
Ignore error codes returned by commands	**-i**
Redirect messages to standard out instead of standard error	**-w**
Show extended error messages	**-x**

Table 7-2: Overview of control program options

For a complete list and description of all control program options, see section 5.5, *Make Utility Options*, in Chapter *Tool Options* of the *Reference Manual*.

7.3.2 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 on a UNIX system this file is not found, the file **`Makefile`** is used as the default. If you want to use other makefiles, use the option **-f** *my_makefile*.

The makefile can contain a mixture of:

- targets and dependencies
- rules
- macro definitions or functions
- 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
```

Targets and dependencies

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

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

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

```
all:                    demo.elf final.elf

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

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

```
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 build. The target `all` depends on `demo.elf` and `final.elf` so the second and third invocation have also the same effect and the files `demo.elf` and `final.elf` are built.

In MS-Windows you can normally use colons to denote drive letters. The following works as intended: `c:foo.o : a:foo.c`

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
```

For target lines, macros and functions are expanded at the moment they are read by the make utility. Normally macros are not expanded until the moment they are actually used.

Special Targets

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

.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.

.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 `mkpcp.mk`.

If you specify this target with dependencies, these are added to the existing `.SUFFIXES` target in `mkpcp.mk`. If you specify this target without dependencies, the existing list is cleared.

.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 **–p** command line option to make all target files precious.

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
            mv 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 (ERRORLEVEL in MS-DOS). 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 COMMAND.COM to execute the command. If the '+' is not followed by a shell line, but the command is a DOS command or if redirection is used (<, |, >), the shell line is passed to COMMAND.COM anyway. For UNIX, redirection, backquote (‘) parentheses and variables force the use of a shell.

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

```
cd c:\cpcp\bin ;\
ccpcp -V
```

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.

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.

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**.

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.src   :
            cpcp $<
```

Read this as: to build a file with extension **.src** out of a file with extension **.c**, call the compiler with **$<**. **$<** 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.

Example

This makefile says that **prog.out** 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**.

```
LIB  =     -lc                                          # 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
```

The following makefile uses implicit rules (from **mkpcp.mk**) to perform the same job.

```
LKFLAGS = -lc               # 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
```

Files

makefile	Description of dependencies and rules.
Makefile	Alternative to makefile, for UNIX.
mkpcp.mk	Default dependencies and rules.

Diagnostics

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

Macro definitions

A macro is a symbol name that is replaced with it's 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 and more text
```

Spaces around the equal sign are not significant. To use a macro, you must access it's 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.

Predefined Macros

MAKE — Holds the value `mkpcp`. Any line which uses `MAKE`, temporarily overrides the option **-n** (Show commands without executing), just for the duration of the one line. This way you can test nested calls to `MAKE` with the option **-n**.

MAKEFLAGS — Holds the set of options provided to **mkpcp** (except for the options **-f** and **-d**). If this macro is exported to set the environment variable `MAKEFLAGS`, the set of options is processed before any command line options. You can pass this macro explicitly to nested **mkpcp**'s, but it is also available to these invocations as an environment variable.

PRODDIR — Holds the name of the directory where **mkpcp** is installed. You can use this macro to refer to files belonging to the product, for example a library source file.

```
DOPRINT = $(PRODDIR)/lib/src/_doprint.c
```

When **mkpcp** is installed in the directory `/cpcp/bin` this line expands to:

```
DOPRINT = /cpcp/lib/src/_doprint.c
```

SHELLCMD Holds the default list of commands which are local to the SHELL. If a rule is an invocation of one of these commands, a SHELL is automatically spawned to handle it.

TMP_CCPROG
Holds the name of the control program: `ccpcp`. If this macro and the TMP_CCOPT macro are set and the command line argument list for the control program exceeds 127 characters, then **mkpcp** creates a temporary file with the command line arguments. **mkpcp** calls the control program with the temporary file as command input file.

TMP_CCOPT
Holds **–f**, the control program option that tells it to read options from a file. (This macro is only available for the Windows command prompt version of **mkpcp**.)

$ This macro translates to a dollar sign. Thus you can use "$$" in the makefile to represent a single "$".

There are several dynamically maintained macros that are useful as abbreviations within rules. It is best not to define them explicitly.

$* The basename of the current target.

$< The name of the current dependency file.

$@ The name of the current target.

$? The names of dependents which are younger than the target.

$! The names of all dependents.

The $< and $* macros are normally used for implicit rules. They may be unreliable when used within explicit target command lines. All macros may be suffixed with F to specify the Filename components (e.g. ${*F}, ${@F}). Likewise, the macros $*, $< and $@ may be suffixed by D to specify the directory component.

The result of the $* macro is always without double quotes (”), regardless of the original target having double quotes (”) around it or not.
The result of using the suffix F (Filename component) or D (Directory component) is also always without double quotes (”), regardless of the original contents having double quotes (”) around it or not.

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)
```

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 **if** 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 using 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.

See also *Defining Macros* in section 5.5, *Make Utility Options*, in Chapter *Tools Options* of the *Reference Manual*.

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
            cpcp 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. Include files may be nested.

```
include makefile2
```

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 macros is exported at the moment the export line is read so the macro definition has to precede the export line.

7.4 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 functionalities:

- 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:

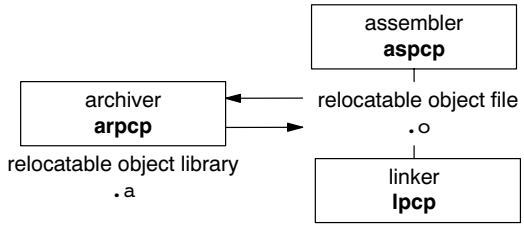

Figure 7-1: ***arpcp*** *ELF/DWARF archiver and library maintainer*

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.

7.4.1 CALLING THE ARCHIVER

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

arpcp *key_option* [sub_*option*...] *library* [*object_file*]

key_option With a key option you specify the main task which the archiver should perform. You must *always* specify a key option.

sub_option Sub-options specify into more detail how the archiver should perform the task that is specified with the key option. It is not obligatory to specify sub-options.

library The name of the library file on which the archiver performs the specified action. You must always specify a library name, except for the option **–?** and **–V**. When the library is not in the current directory, specify the complete path (either absolute or relative) to the library.

object_file The name of an object file. You must always specify an object file name when you add, extract, replace or remove an object file from the library.

Options of the archiver utility

The following archiver options are available:

Description	Option	Sub–option
Main functions (key options)		
Replace or add an object module	**–r**	**–a –b –c –u –v**
Extract an object module from the library	**–x**	**–v**
Delete object module from library	**–d**	**–v**
Move object module to another position	**–m**	**–a –b –v**
Print a table of contents of the library	**–t**	**–s0 –s1**
Print object module to standard output	**–p**	
Sub–options		
Append or move new modules after existing module *name*	**–a** *name*	
Append or move new modules before existing module *name*	**–b** *name*	
Create library without notification if library does not 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**	

Description	Option	Sub–option
Miscellaneous		
Display options	**–?**	
Display version header	**–V**	
Read options from *file*	**–f** *file*	
Suppress warnings above level *n*	**–w***n*	

Table 7-3: Overview of archiver options and sub-options

For a complete list and description of all archiver options, see section 5.6, *Archiver Options*, in Chapter *Tool Options* of the *Reference Manual*.

7.4.2 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
```

INDEX

INDEX

TASKING

INDEX

Symbols

A

B

C

D

E

F

H

I

J

L

M

N

O

P

Q

R

S

T

U

V