Chipsmall Limited consists of a professional team with an average of over 10 year of expertise in the distribution of electronic components. Based in Hongkong, we have already established firm and mutual-benefit business relationships with customers from,Europe,America and south Asia,supplying obsolete and hard-to-find components to meet their specific needs.

With the principle of “Quality Parts,Customers Priority,Honest Operation,and Considerate Service”,our business mainly focus on the distribution of electronic components. Line cards we deal with include Microchip,ALPS,ROHM,Xilinx,Pulse,ON,Everlight and Freescale. Main products comprise IC,Modules,Potentiometer,IC Socket,Relay,Connector.Our parts cover such applications as commercial,industrial, and automotives areas.

We are looking forward to setting up business relationship with you and hope to provide you with the best service and solution. Let us make a better world for our industry!

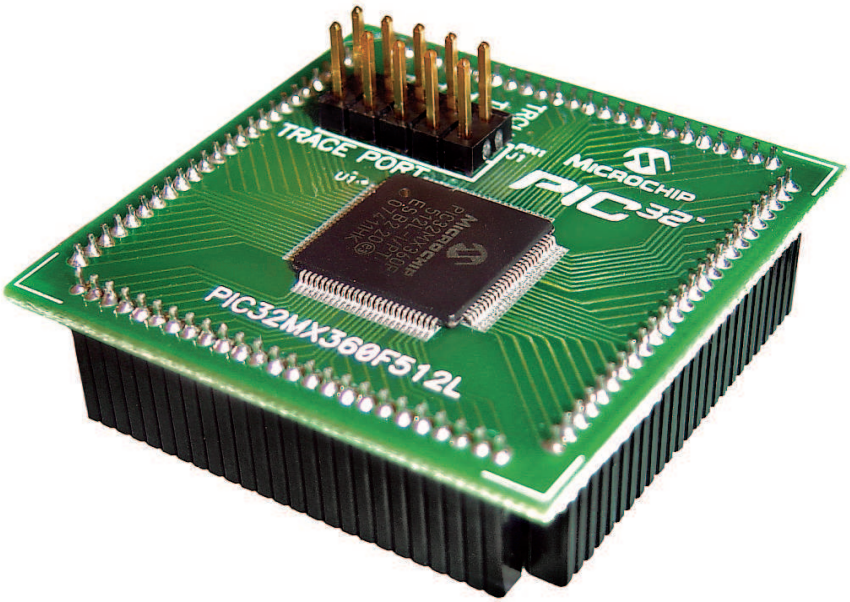
Contact us

Tel: +86-755-8981 8866 Fax: +86-755-8427 6832
Email & Skype: info@chipsmall.com Web: www.chipsmall.com
Address: A1208, Overseas Decoration Building, #122 Zhenhua RD., Futian, Shenzhen, China

HI-TECH SOFTWARE

Manual

HI-TECH C® Tools for the PIC32 MCU Family



HI-TECH C TOOLS for the PIC32 MCU Family

HI-TECH Software

Copyright (C) 2008 HI-TECH Software.
All Rights Reserved. Printed in Australia.
Produced on: July 22, 2008

HI-TECH Software Pty. Ltd.
ACN 002 724 549
45 Colebard Street West
Acacia Ridge QLD 4110
Australia

email: hitech@htsoft.com
web: http://microchip.htsoft.com
ftp: ftp://www.htsoft.com

Contents

List of Tables

Chapter 1

Introduction

1.1 Typographic conventions

Different fonts and styles are used throughout this manual to indicate special words or text. Computer prompts, responses and filenames will be printed in `constant-spaced` type**.** When the filename is the name of a standard header file, the name will be enclosed in angle brackets, e.g. `<stdio.h>`. These header files can be found in the `INCLUDE` directory of your distribution.

Samples of code, C keywords or types, assembler instructions and labels will also be printed in a `constant-space` type. Assembler code is printed in a font similar to that used by C code.

Particularly useful points and new terms will be emphasized using *italicized type*. When part of a term requires substitution, that part should be printed in the appropriate font, but in *italics*. For example: `#include <`*`filename.h`*`>`.

Chapter 2

PICC32 Command-line Driver

`PICC32` is the driver invoked from the command line to perform all aspects of compilation, including C code generation, assembly and link steps. It is the recommended way to use the compiler as it hides the complexity of all the internal applications used in the compilation process and provides a consistent interface for all compilation steps.

This chapter describes the steps the driver takes during compilation, files that the driver can accept and produce, as well as the command-line options that control the compiler's operation.

WHAT IS "THE COMPILER"? Throughout this manual, the term "the compiler" is used to refer to either all, or some subset of, the collection of applications that form the HI-TECH C PRO for the PIC32 MCU Family package. Often it is not important to know, for example, whether an action is performed by the parser or code generator application, and it is sufficient to say it was performed by "the compiler".

It is also reasonable for "the compiler" to refer to the command-line driver (or just "driver"), PICC32 , as this is the application executed to invoke the compilation process. Following this view, "compiler options" should be considered command-line driver options, unless otherwise specified in this manual.

Similarly "compilation" refers to all, or some part of, the steps involved in generating source code into an executable binary image.

Table 2.1: PICC32 input file types

File Type	Meaning
`.c`	C source file
`.p1`	p-code file
`.lpp`	p-code library file
`.as`	Assembler source file
`.obj`	Relocatable object code file
`.lib`	Relocatable object library file
`.hex`	Intel HEX file

2.1 Invoking the Compiler

This chapter looks at how to use PICC32 as well as the tasks that it and the internal applications perform during compilation.

`PICC32` has the following basic command format:

```
PICC32 [options] files [libraries]
```

It is conventional to supply *`options`*(identified by a leading *dash* "-" or *double dash* "–") before the filenames, although this is not mandatory.

The formats of the options are discussed below in Section 2.6, and a detailed description of each option follows.

The *`files`* may be any mixture of C and assembler source files, and precompiled intermediate files, such as relocatable object (`.obj`) files or p-code (`.p1`) files. The order of the files is not important, except that it may affect the order in which code or data appears in memory, and may affect the name of some of the output files.

`Libraries` is a list of either object code or p-code library files that will be searched by the linker.

`PICC32` distinguishes source files, intermediate files and library files solely by the *file type* or *extension*. Recognized file types are listed in Table 2.1. This means, for example, that an assembler file must always have a `.as` extension. Alphabetic case of the extension is not important from the compiler's point of view.

MODULES AND SOURCE FILES: A C *source file* is a file on disk that contains all or part of a program. C source files are initially passed to the preprocessor by the driver.
A *module* is the output of the preprocessor, for a given source file, after inclusion of any header files (or other source files) which are specified by `#include` preprocessor

directives. These modules are then passed to the remainder of the compiler applications. Thus, a module may consist of several source and header files. A module is also often referred to as a *translation unit*. These terms can also be applied to assembly files, as they too can include other header and source files.

Some of the compiler's output files contain project-wide information and are not directly associated with any one particular input file, e.g. the map file. If the names of these project-wide files are not specified on the command line, the basename of these files is derived from the first C source file listed on the command line. If there are no files of this type being compiled, the name is based on the first input file (regardless of type) on the command line. Throughout this manual, the basename of this file will be called the *project name*.

Most IDEs use project files whose names are user-specified. Typically the names of project-wide files, such as map files, are named after the project, however check the manual for the IDE you are using for more details.

2.1.1 Long Command Lines

The PICC32 driver is capable of processing command lines exceeding any operating system limitation. To do this, the driver may be passed options via a command file. The command file is read by using the `@` symbol which should be immediately followed (i.e. no intermediate space character) by the name of the file containing the command line arguments.

The file may contain blank lines, which are simply skipped by the driver. The command-line arguments may be placed over several lines by using a *space* and *backslash* character for all non-blank lines, except for the last line.

The use of a command file means that compiler options and project filenames can be stored along with the project, making them more easily accessible and permanently recorded for future use.

TUTORIAL

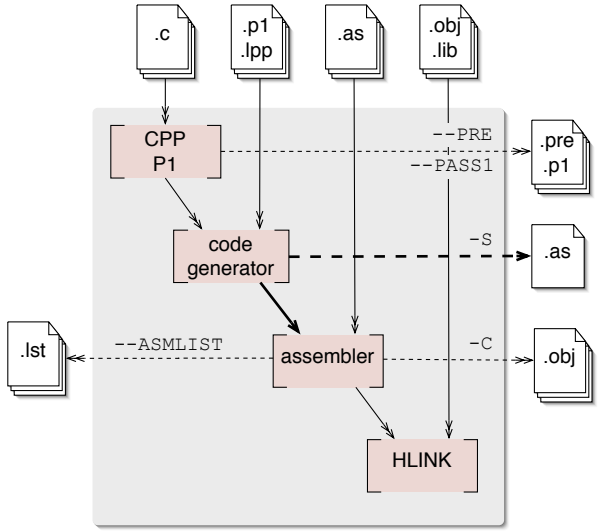
USING COMMAND FILES A command file `xyz.cmd` is constructed with your favorite text editor and contains both the options and file names that are required to compile your project as follows:

```
--chip=32MX360F512L -m \
--opt=all -g \
main.c isr.c
```

After it is saved, the compiler may be invoked with the command:

```
PICC32  @xyz.cmd
```

Figure 2.1: Flow diagram of the initial compilation sequence



2.2 The Compilation Sequence

`PICC32` will check each file argument and perform appropriate actions on each file. The entire compilation sequence can be thought of as the initial sequence up to the link stage, and the final sequence which takes in the link step and any post link steps required.

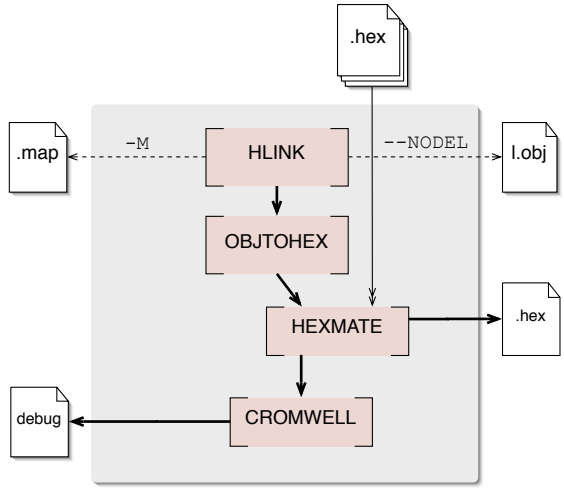
Graphically the compilation steps up to the link stage are illustrated in Figure 2.1. This diagram shows all possible input files along the top; intermediate and transitional files, along the right side; and useful compiler output files along the left. Generated files are shown along with the options that are used to generate and preserve these. All the files shown on the right, can be generated and fed to the compiler in a subsequent compile step; those on the left are used for debug purposes and cannot be used as an input to any subsequent compilation.

The individual compiler applications are shown as boxes. The C preprocessor, `CPP`, and parser, `P1`, have been grouped together for clarity.

The thin, multi-arrowed lines indicate the flow of multiple files — one for each file being processed by the revel ant application. The thick single-arrowed lines indicate a single file for the project being compiled. Thus, for example, when using the `--PASS1` driver option, the parser produces one `.p1` file for each C source file that is being compiled as part of the project, but the code generator produces only one `.as` file from all `.c`, `.p1` and `.lpp` input files which it is passed.

Dotted lines indicate a process that may require an option to create or preserve the indicated file.

Figure 2.2: Flow diagram of the final compilation sequence



The link and post-link steps are graphically illustrated in Figure 2.2.

This diagram shows `.hex` files as additional input file type not considered in the initial compilation sequence. These files can be merged into the `.hex` file generated from the other input files in the project by an application called `HEXMATE`. See Section 5.14 for more information on this utility.

The output of the linker is a single absolute object file, called `l.obj`, that can be preserved by using the `--NODEL` driver option. Without this option, this temporary file is used to generate an output file (e.g. a HEX file) and files used for debugging by development tools (e.g. COFF files) before it is deleted. The file `l.obj` can be used as the input to `OBJTOHEX` if running this application manually, but it cannot be passed to the driver as an input file as it absolute and cannot be further processed.

2.2.1 Single-step Compilation

The command-line driver, PICC32 , can compile any mix of input files in a single step. All source files will be re-compiled regardless of whether they have been changes since that last time a compilation was performed.

Unless otherwise specified, a default output file and debug file are produced. All intermediate files (`.p1` and `.obj`) remain after compilation has completed, but all other transitional files are deleted, unless you use the `--NODEL` option which preserves all generated files. Note some generated files may be in a temporary directory not associated with your project and use a pseudo-randomly

generated filename.

TUTORIAL

SINGLE STEP COMPILATION The files, `main.c`, `io.c`, `mdef.as`, `sprt.obj`, `a_sb.lib` and `c_sb.lpp` are to be compiled. To perform this in a single step, the following command line can be used as a starting point for the project development.

```
PICC32  --chip=32MX360F512L main.c io.c mdef.as sprt.obj a_sb.lib c_sb.lpp
```

This will run the C pre-processor then the parser with `main.c` as input, and then again for `io.c` producing two p-code files. These two files, in addition to the library file `c_sb.lpp`, are passed to the code generator producing a single temporary assembler file output. The assembler is then executed and is passed the output of the code generator. It is run again with `mdef.as`, producing two relocatable object files. The linker is then executed, passing in the assembler output files in addition to `sprt.obj` and the library file `a_sb.lib`. The output is a single absolute object file, `l.obj`. This is then passed to the appropriate post-link utility applications to generate the specified output file format and debugging files. All temporary files, including `l.obj`, are then deleted. The intermediate files: p-code and relocatable object files, are not deleted. This tutorial does not consider the runtime startup code that is automatically generated by the driver.

2.2.2 Generating Intermediate Files

The HI-TECH C PRO for the PIC32 MCU Family version compiler uses two types of intermediate files. For C source files, the p-code file (`.p1` file) is used as the intermediate file. For assembler source files, the relocatable object file (`.obj` file) is used.

You may wish to generate intermediate files for several reasons, but the most likely will be if you are using an IDE or make system that allows an incremental build of the project. The advantage of a incremental build is that only the source files that have been modified since the last build need to be recompiled before again running the final link step. This dependency checking may result in reduced compilation times, particularly if there are a large number of source files.

You may also wish to generate intermediate files to construct your own library files, although PICC32 is capable of constructing libraries in a single step, so this is typically not necessary. See Section 2.6.42 for more information.

Intermediate files may also assist with debugging a project that fails to work as expected.

If a multi-step compilation is required the recommended compile sequence is as follows.

- Compile all modified C source files to p-code files using the `--PASS1` driver option

- Compile all modified assembler source files to relocatable object files using the `-C` driver option
- Compile all p-code and relocatable object files into a single output object file

The final step not only involves the link stage, but also code generation of all the p-code files. In effect, the HI-TECH C PRO for the PIC32 MCU Family version code generator performs some of the tasks normally performed by the linker. Any user-specified (non standard) libraries also need to be passed to the compiler during the final step. This is the incremental build sequence used by HI-TIDE™.

TUTORIAL

MULTI-STEP COMPILATION The files in the previous example are to be compiled using a multi-step compilation. The following could be used.

```
PICC32  --chip=32MX360F512L --pass1 main.c
PICC32  --chip=32MX360F512L --pass1 io.c
PICC32  --chip=32MX360F512L -c mdef.as
PICC32  --chip=32MX360F512L main.p1 io.p1 mdef.obj sprt.obj c_sb.lpp a_sb.lib
```

If using a make system with incremental builds, only those source files that have changed since the last build need the first compilation step performed again, so not all of the first three steps need be executed.

If is important to note that the code generator needs to compile all p-code or p-code library files in the one step. Thus, if the `--PASS1` option is not used (or `--PRE` is not used), all C source files, and any p-code libraries, must be built together in the one command.

If a compilation is performed, and the source file that contains `main()` is not present in the list of C source files, an undefined symbol error for `_main` will be produced by the code generator. If the file that contains the definition for `main()` is present, but it is a subset of the C source files making up a project that is being compiled, the code generator will not be able to see the entire C program and this will defeat most of the optimization techniques employed by the code generator.

There may be multi-step compilation methods employed that lead to compiler errors as a result of the above restrictions, for example you cannot have an C function compiled into a p-code library that is called only from assembler code.

2.2.3 Special Processing

There are several special steps that take place during compilation.