



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

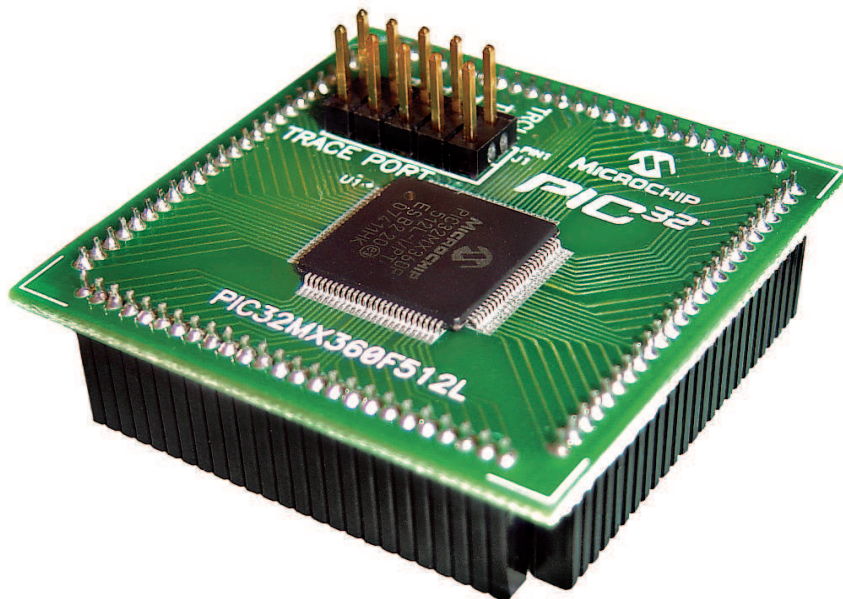
Email & Skype: info@chipsmall.com Web: www.chipsmall.com

Address: A1208, Overseas Decoration Building, #122 Zhenhua RD., Futian, Shenzhen, China





HI-TECH 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

Table of Contents	3
List of Tables	15
1 Introduction	17
1.1 Typographic conventions	17
2 PICC32 Command-line Driver	19
2.1 Invoking the Compiler	20
2.1.1 Long Command Lines	21
2.2 The Compilation Sequence	22
2.2.1 Single-step Compilation	23
2.2.2 Generating Intermediate Files	24
2.2.3 Special Processing	25
2.2.3.1 Printf check	26
2.2.3.2 Assembly Code Requirements	26
2.3 Runtime Files	26
2.3.1 Library Files	27
2.3.1.1 Standard Libraries	28
2.3.1.2 Peripheral Libraries	28
2.3.2 Runtime Startup Module	28
2.3.2.1 Initialization of Data psects	29
2.3.2.2 Clearing the Bss Psect	30
2.3.2.3 System Coprocessor Initialization (--RUNTIME=cp0)	31
2.3.2.4 General Purpose Register Intialization (--RUNTIME=gpr)	31
2.3.2.5 The Stack and Heap (--RUNTIME=stack, --RUNTIME=heap)	31
2.3.2.6 Default Configuration Words (--RUNTIME=config)	32
2.3.2.7 Memory Performance (--RUNTIME=perform)	32

2.3.3	The Powerup Routine	33
2.3.4	The <code>printf</code> Routine	33
2.4	Debugging Information	35
2.4.1	Output File Formats	35
2.4.2	Symbol Files	35
2.5	Compiler Messages	36
2.5.1	Messaging Overview	36
2.5.2	Message Language	37
2.5.3	Message Type	37
2.5.4	Message Format	38
2.5.5	Changing Message Behaviour	40
2.5.5.1	Disabling Messages	41
2.5.5.2	Changing Message Types	41
2.6	PICC32 Driver Option Descriptions	41
2.6.1	<code>-C</code> : Compile to Object File	42
2.6.2	<code>-Dmacro</code> : Define Macro	42
2.6.3	<code>-Efile</code> : Redirect Compiler Errors to a File	43
2.6.4	<code>-Gfile</code> : Generate Source-level Symbol File	44
2.6.5	<code>-Ipath</code> : Include Search Path	44
2.6.6	<code>-Llibrary</code> : Scan Library	45
2.6.7	<code>-Loption</code> : Adjust Linker Options Directly	45
2.6.8	<code>-Mfile</code> : Generate Map File	47
2.6.9	<code>-Nsize</code> : Identifier Length	47
2.6.10	<code>-Ofile</code> : Specify Output File	47
2.6.11	<code>-P</code> : Preprocess Assembly Files	48
2.6.12	<code>-Q</code> : Quiet Mode	48
2.6.13	<code>-S</code> : Compile to Assembler Code	48
2.6.14	<code>-Umacro</code> : Undefine a Macro	48
2.6.15	<code>-V</code> : Verbose Compile	49
2.6.16	<code>-X</code> : Strip Local Symbols	49
2.6.17	<code>--ASMLIST</code> : Generate Assembler <code>.LST</code> Files	49
2.6.18	<code>--CALLGRAPH=type</code> : Select callgraph type	49
2.6.19	<code>--CHECKSUM=start-end@destination<,specs></code> : Calculate a checksum	50
2.6.20	<code>--CHIP=processor</code> : Define Processor	50
2.6.21	<code>--CHIPINFO</code> : Display List of Supported Processors	50
2.6.22	<code>--CR=file</code> : Generate Cross Reference Listing	50
2.6.23	<code>--DEBUGGER=type</code> : Select Debugger Type	51
2.6.24	<code>--ECHO</code> : Echo command line before processing	51

2.6.25	--ERRFORMAT= <i>format</i> : Define Format for Compiler Messages	51
2.6.26	--ERRORS= <i>number</i> : Maximum Number of Errors	51
2.6.27	--FILL= <i>opcode</i> : Fill Unused Program Memory	52
2.6.28	--GETOPTION= <i>app, file</i> : Get Command-line Options	52
2.6.29	--HELP[=< <i>option</i> >]: Display Help	52
2.6.30	--IDE= <i>type</i> : Specify the IDE being used	52
2.6.31	--INTERRUPTS= <i>suboption, <suboption></i> : Specify the Interrupts Scheme	52
2.6.32	--ISA=< <i>type</i> >: Specify the Instruction Set Architecture	54
2.6.33	--LANG= <i>language</i> : Specify the Language for Messages	54
2.6.34	--MEMMAP= <i>file</i> : Display Memory Map	55
2.6.35	--MSGDISABLE= <i>messagelist</i> : Disable Warning Messages	55
2.6.36	--MSGFORMAT= <i>format</i> : Set Advisory Message Format	55
2.6.37	--NODEL: Do not remove temporary files	55
2.6.38	--NOEXEC: Don't Execute Compiler	55
2.6.39	--OBJDIR= <i>path</i> : Specify a directory for Object files	55
2.6.40	--OPT[=< <i>type</i> >]: Invoke Compiler Optimizations	56
2.6.41	--OUTDIR= <i>path</i> : Specify a directory for Output files	56
2.6.42	--OUTPUT= <i>type</i> : Specify Output File Type	56
2.6.43	--PASS1: Compile to P-code	57
2.6.44	--PRE: Produce Preprocessed Source Code	57
2.6.45	--PROTO: Generate Prototypes	58
2.6.46	--RAM= <i>lo-hi, <lo-hi, ...></i> : Specify Additional RAM Ranges	59
2.6.47	--ROM= <i>lo-hi, <lo-hi, ...> tag</i> : Specify Additional ROM Ranges	59
2.6.48	--RUNTIME= <i>type</i> : Specify Runtime Environment	60
2.6.49	--SCANDEP: Scan for Dependencies	61
2.6.50	--SERIAL= <i>hexcode@address</i> : Store a Value at this Program Memory Address	61
2.6.51	--SETOPTION= <i>app, file</i> : Set the Command-line Options for Application	62
2.6.52	--STRICT: Strict ANSI Conformance	62
2.6.53	--STRICTCALLS: Strict MIPS Parameter Passing	62
2.6.54	--SUMMARY= <i>type</i> : Select Memory Summary Output Type	63
2.6.55	--TIME: Report time taken for each phase of build process	63
2.6.56	--VER: Display The Compiler's Version Information	63
2.6.57	--WARN= <i>level</i> : Set Warning Level	63
2.6.58	--WARNFORMAT= <i>format</i> : Set Warning Message Format	64

3	C Language Features	65
3.1	ANSI Standard Issues	65
3.1.1	Divergence from the ANSI C Standard	65
3.1.2	Implementation-defined behaviour	65
3.1.3	Non-ANSI Operations	65
3.2	Processor-related Features	66
3.2.1	Processor Support	66
3.2.2	Configuration Fuses	66
3.2.3	Multi-byte SFRs	67
3.3	Supported Data Types and Variables	67
3.3.1	Radix Specifiers and Constants	67
3.3.2	Bit Data Types and Variables	69
3.3.3	Using Bit-Addressable Registers	70
3.3.4	8-Bit Integer Data Types and Variables	71
3.3.5	16-Bit Integer Data Types	71
3.3.6	32-Bit Integer Data Types and Variables	72
3.3.7	Floating Point Types and Variables	72
3.3.8	Structures and Unions	73
3.3.8.1	Bit-fields in Structures	73
3.3.8.2	Structure and Union Qualifiers	74
3.3.9	Standard Type Qualifiers	75
3.3.9.1	Const and Volatile Type Qualifiers	75
3.3.10	Special Type Qualifiers	76
3.3.10.1	Persistent Type Qualifier	76
3.3.10.2	cp0 Type Qualifier	77
3.3.10.3	sfr Type Qualifier	77
3.3.10.4	__strictcall Function Qualifier	78
3.3.10.5	ISA Function Qualifiers	78
3.3.11	Pointer Types	78
3.3.11.1	Pointers to Const	78
3.3.11.2	Function Pointers	79
3.4	Storage Class and Object Placement	79
3.4.1	Local Variables	79
3.4.1.1	Auto Variables	79
3.4.1.2	Static Variables	79
3.4.2	Absolute Variables	80
3.4.3	Objects in Program Space	80
3.5	Functions	81
3.5.1	Absolute Functions	81

3.5.2	Function Argument Passing	81
3.5.3	Function Return Values	81
3.5.4	Function Stack Frame	82
3.6	Register Usage Conventions	82
3.7	Operators	83
3.7.1	Integral Promotion	83
3.7.2	Shifts applied to integral types	84
3.7.3	Division and modulus with integral types	85
3.8	Psects	85
3.8.1	Compiler-generated Psects	85
3.9	Interrupt Handling in C	87
3.9.1	Interrupt Functions	87
3.9.2	Interrupt & Exception Types	87
3.9.3	Runtime Startup Callback Sequence	88
3.10	Mixing C and Assembly Code	88
3.10.1	External Assembly Language Functions	89
3.10.2	#asm, #endasm and asm()	92
3.10.3	Accessing C objects from within Assembly Code	93
3.10.3.1	Accessing special function register names from assembler	94
3.10.4	Interaction between Assembly and C Code	94
3.10.4.1	Absolute Psects	94
3.10.4.2	Undefined Symbols	95
3.11	Preprocessing	96
3.11.1	C Language Comments	96
3.11.2	Preprocessor Directives	97
3.11.3	Predefined Macros	97
3.11.4	Pragma Directives	100
3.11.4.1	The #pragma jis and nojis Directives	100
3.11.4.2	The #pragma printf_check Directive	100
3.11.4.3	The #pragma regsused Directive	101
3.11.4.4	The #pragma switch Directive	101
3.11.4.5	The #pragma warning Directive	102
3.12	Linking Programs	104
3.12.1	Replacing Library Modules	105
3.12.2	Signature Checking	105
3.12.3	Linker-Defined Symbols	107
3.13	Standard I/O Functions and Serial I/O	107

4	Macro Assembler	109
4.1	Assembler Usage	109
4.2	Assembler Options	110
4.3	HI-TECH C Assembly Language	113
4.3.1	Pre-defined Macros	113
4.3.1.1	MIPS32r2 In-built Macro Instructions	113
4.3.1.2	MIPS16E In-built Macro Instructions	113
4.3.2	Instruction Operand Variants	114
4.3.3	Statement Formats	114
4.3.4	Characters	114
4.3.4.1	Delimiters	114
4.3.4.2	Special Characters	114
4.3.5	Comments	114
4.3.5.1	Special Comment Strings	116
4.3.6	Constants	116
4.3.6.1	Numeric Constants	116
4.3.6.2	Character Constants and Strings	117
4.3.7	Identifiers	117
4.3.7.1	Significance of Identifiers	117
4.3.7.2	Assembler-Generated Identifiers	118
4.3.7.3	Symbolic Labels	118
4.3.8	Expressions	118
4.3.9	Program Sections	120
4.3.10	Assembler Directives	121
4.3.10.1	GLOBAL	121
4.3.10.2	END	121
4.3.10.3	PSECT	121
4.3.10.4	ORG	124
4.3.10.5	EQU	125
4.3.10.6	SET	126
4.3.10.7	DB	126
4.3.10.8	DH	126
4.3.10.9	DSTR	126
4.3.10.10	DSTRZ	126
4.3.10.11	DW	126
4.3.10.12	DS	127
4.3.10.13	DABS	127
4.3.10.14	IF, ELSIF, ELSE and ENDIF	127
4.3.10.15	MACRO and ENDM	128

4.3.10.16	LOCAL	129
4.3.10.17	ALIGN	130
4.3.10.18	REPT	130
4.3.10.19	IRP and IRPC	130
4.3.10.20	PROCESSOR	131
4.3.10.21	SIGNAT	131
4.3.11	Assembler Controls	132
4.3.11.1	ASMOPT_ON	132
4.3.11.2	ASMOPT_OFF	133
4.3.11.3	FORCE_EXTEND	133
4.3.11.4	NEVER_EXTEND	133
4.3.11.5	AUTO_EXTEND	133
4.3.11.6	COND	133
4.3.11.7	EXPAND	133
4.3.11.8	INCLUDE	133
4.3.11.9	LIST	134
4.3.11.10	NOCOND	134
4.3.11.11	NOEXPAND	134
4.3.11.12	NOLIST	134
4.3.11.13	NOXREF	134
4.3.11.14	PAGE	135
4.3.11.15	SPACE	135
4.3.11.16	SUBTITLE	135
4.3.11.17	TITLE	135
4.3.11.18	XREF	135
5	Linker and Utilities	137
5.1	Introduction	137
5.2	Relocation and Psects	137
5.3	Program Sections	138
5.4	Local Psects	138
5.5	Global Symbols	138
5.6	Link and load addresses	139
5.7	Operation	139
5.7.1	Numbers in linker options	140
5.7.2	-Aclass=low-high,...	141
5.7.3	-Cx	141
5.7.4	-Cpsect=class	141
5.7.5	-Dclass=delta	142

5.7.6	-Dsymfile	142
5.7.7	-Eerrfile	142
5.7.8	-F	142
5.7.9	-Gspec	142
5.7.10	-Hsymfile	143
5.7.11	-H+symfile	143
5.7.12	-Jerrcount	143
5.7.13	-K	143
5.7.14	-I	144
5.7.15	-L	144
5.7.16	-LM	144
5.7.17	-Mmapfile	144
5.7.18	-N, -Ns and -Nc	144
5.7.19	-Ooutfile	144
5.7.20	-Pspec	144
5.7.21	-Qprocessor	146
5.7.22	-S	146
5.7.23	-Sclass=limit[, bound]	146
5.7.24	-Usymbol	147
5.7.25	-Vavmap	147
5.7.26	-Wnum	147
5.7.27	-X	147
5.7.28	-Z	147
5.8	Invoking the Linker	148
5.9	Map Files	148
5.9.1	Generation	148
5.9.2	Contents	149
5.9.2.1	General Information	149
5.9.2.2	Psect Information listed by Module	151
5.9.2.3	Psect Information listed by Class	153
5.9.2.4	Segment Listing	153
5.9.2.5	Unused Address Ranges	153
5.9.2.6	Symbol Table	154
5.10	Librarian	155
5.10.1	The Library Format	155
5.10.2	Using the Librarian	156
5.10.3	Examples	157
5.10.4	Supplying Arguments	157
5.10.5	Listing Format	157

5.10.6	Ordering of Libraries	158
5.10.7	Error Messages	158
5.11	Objtohex	158
5.11.1	Checksum Specifications	158
5.12	Cref	160
5.12.1	-Fprefix	160
5.12.2	-Hheading	161
5.12.3	-Llen	161
5.12.4	-Ooutfile	161
5.12.5	-Pwidth	161
5.12.6	-Sstoplist	161
5.12.7	-Xprefix	162
5.13	Cromwell	162
5.13.1	-Pname[,architecture]	163
5.13.2	-N	163
5.13.3	-D	163
5.13.4	-C	164
5.13.5	-F	164
5.13.6	-Okey	164
5.13.7	-Ikey	164
5.13.8	-L	164
5.13.9	-E	165
5.13.10	-B	165
5.13.11	-M	165
5.13.12	-V	165
5.14	Hexmate	165
5.14.1	Hexmate Command Line Options	166
5.14.1.1	specifications,filename.hex	166
5.14.1.2	+ Prefix	168
5.14.1.3	-ADDRESSING	168
5.14.1.4	-BREAK	169
5.14.1.5	-CK	169
5.14.1.6	-FILL	170
5.14.1.7	-FIND	171
5.14.1.8	-FIND...,DELETE	172
5.14.1.9	-FIND...,REPLACE	172
5.14.1.10	-FORMAT	172
5.14.1.11	-HELP	173
5.14.1.12	-LOGFILE	173

5.14.1.13 -MASK	174
5.14.1.14 -Ofile	174
5.14.1.15 -SERIAL	174
5.14.1.16 -SIZE	175
5.14.1.17 -STRING	175
5.14.1.18 -STRPACK	176
A Library Functions	177
__CONFIG	178
ABS	179
ACOS	180
ASCTIME	181
ASIN	183
ASSERT	184
ATAN	185
ATAN2	186
ATOF	187
ATOI	188
ATOL	189
BSEARCH	190
CEIL	192
CGETS	193
COS	195
COSH	196
CPUTS	197
CTIME	198
DIV	199
EVAL_POLY	200
EXP	201
FABS	202
FMOD	203
FLOOR	204
FREXP	205
FTOA	206
GETCH	207
GETCHAR	208
GETS	209
GMTIME	210
ISALNUM	212

ISDIG	214
ITOA	215
LABS	216
LDEXP	217
LDIV	218
LOCALTIME	219
LOG	221
LONGJMP	222
LTOA	224
MEMCMP	225
MKTIME	227
MODF	229
POW	232
PUTCH	233
PUTCHAR	234
PUTS	236
QSORT	237
RAND	239
ROUND	241
SETJMP	244
SETVECTOR	246
SIN	247
SQRT	248
SRAND	249
STRCAT	250
STRCHR	251
STRCMP	253
STRCPY	255
STRCSPN	256
STRLEN	257
STRNCAT	258
STRNCMP	260
STRNCPY	262
STRPBRK	264
STRRCHR	265
STRSPN	266
STRSTR	267
STRTOD	268
STRTOL	270

STRTok	272
TAN	274
TIME	275
TOLOWER	277
TRUNC	278
UDIV	279
ULDIV	280
UNGETCH	281
UTOA	282
VA_START	283
XTOI	285
B Error and Warning Messages	287
1...	287
138...	295
184...	301
227...	309
269...	318
312...	324
355...	332
400...	341
444...	346
489...	354
599...	361
670...	365
730...	370
776...	379
836...	384
904...	389
970...	394
1031...	399
1157...	404
1242...	409
C Chip Information	415
D Configuration Attributes	417
Index	421

List of Tables

2.1	PIC32 input file types	20
2.2	Default configuration settings	32
2.3	Support languages	37
2.4	Messaging environment variables	39
2.5	Messaging placeholders	39
2.6	Supported IDEs	52
2.7	--interrupts sub-options affecting the type of interrupt vectors.	53
2.8	--interrupts sub-options affecting the location of the vector table.	53
2.9	--interrupts sub-options affecting the number of vectors to service interrupts.	54
2.10	Supported languages	54
2.11	Optimization Options	56
2.12	Output file formats	57
2.14	Memory Summary Suboptions	63
3.1	Basic data types	68
3.2	Radix formats	68
3.3	Floating-point formats	73
3.4	Floating-point format example IEEE 754	73
3.5	Integral division	85
3.6	Preprocessor directives	99
3.8	Pragma directives	100
3.9	Valid register names	102
3.10	Switch types	102
3.11	Supported standard I/O functions	107
4.1	ASPIC32 command-line options	110
4.2	MIPS32r2 Assembly Instruction Operand Variants.	115

4.3	MIPS16e Assembly Instruction Operand Variants	116
4.4	ASPIC32 statement formats	116
4.5	ASPIC32 numbers and bases	117
4.6	ASPIC32 operators	119
4.7	ASPIC32 assembler directives	122
4.8	PSECT flags	123
4.9	psect isa flag suboptions.	125
4.10	PIC32 assembler controls	132
4.11	LIST control options	134
5.1	Linker command-line options	139
5.1	Linker command-line options	140
5.2	Librarian command-line options	156
5.3	Librarian key letter commands	156
5.4	OBJTOHEX command-line options	159
5.5	CREF command-line options	161
5.6	CROMWELL format types	162
5.7	CROMWELL command-line options	163
5.8	-P option architecture arguments for COFF file output.	164
5.9	Hexmate command-line options	167
5.10	Hexmate Checksum Algorithm Selection	170
5.11	INHX types used in -FORMAT option	173
C.1	Devices supported by HI-TECH C PRO for the PIC32 MCU Family	415

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

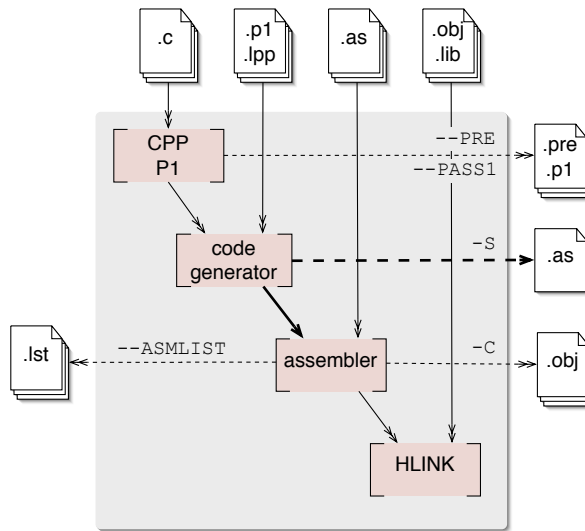
USING COMMAND FILES A command file `xyz.cmd` is constructed with your favorite text editor and contains both the options and file names that are required to compile your project as follows:

```
--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.

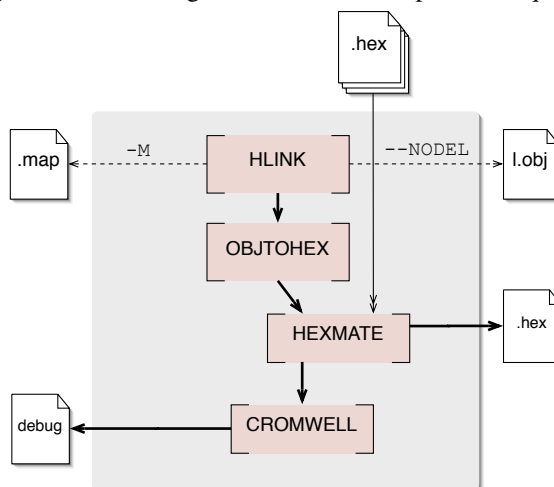
Graphically the compilation steps up to the link stage are illustrated in Figure 2.1. This diagram shows all possible input files along the top; intermediate and transitional files, along the right side; and useful compiler output files along the left. Generated files are shown along with the options 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 relevant 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 (`.pl` 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.pl io.pl 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.

It 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.