

user's
guide

picinit[®]

ANSI C
COMPILER

PICLITE C Manual

Copyright ©1999 HI-TECH Software.
All Rights Reserved. Printed in Australia.

First Printing (c), January 2000

HI-TECH Software
A division of Gretetoy Pty. Ltd. ACN 002 724 549
PO Box 103
Alderley QLD 4051
Australia

Email: hitech@htsoft.com
Web: <http://www.htsoft.com/>
FTP: <ftp.htsoft.com>

Introduction	1
Tutorials	2
Using HTLPIC	3
PICL Command Line Compiler Driver	4
Features and Runtime Environment	5
The PICL Macro Assembler	6
Linker and Utilities Reference Manual	7

1 - Introduction	15
1.1 Compiler Limitations	15
1.2 Typographic conventions	15
1.3 Using This Manual	15
2 - Tutorials	17
2.1 Overview of the compilation process	17
2.1.1 Compilation	17
2.1.2 The compiler input	18
2.1.2.1 Steps before linking	21
2.1.2.2 The link stage	27
2.2 Psects and the linker	29
2.2.1 Psects	29
2.2.1.1 The psect directive	30
2.2.1.2 Psect types	31
2.3 Linking the psects	32
2.3.1 Grouping psects	33
2.3.2 Positioning psects	33
2.3.3 Linker options to position psects	34
2.3.3.1 Placing psects at an address	34
2.3.3.2 Exceptional cases	37
2.3.3.3 Psect classes	38
2.3.3.4 User-defined psects	40
2.3.4 Issues when linking-	41
2.3.4.1 Paged memory	41
2.3.4.2 Separate memory areas	42
2.3.4.3 Objects at absolute addresses	43
2.3.5 Modifying the linker options	44
2.4 Addresses used with the PIC	45
2.4.1 Code addresses	46
2.4.2 Data addresses	46
2.4.3 Bit addresses	49

3 - Using HTLPIC - - - - - 51

3.1 Introduction - - - - -	51
3.1.1 Starting HTLPIC - - - - -	51
3.2 The HI-TECH Windows User Interface - - - - -	52
3.2.1 Environment variables - - - - -	52
3.2.2 Hardware Requirements - - - - -	52
3.2.3 Colours - - - - -	53
3.2.4 Pull-Down Menus - - - - -	53
3.2.4.1 Keyboard Menu Selection - - - - -	54
3.2.4.2 Mouse Menu Selection - - - - -	55
3.2.4.3 Menu Hot Keys - - - - -	56
3.2.5 Selecting windows - - - - -	56
3.2.6 Moving and Resizing Windows - - - - -	58
3.2.7 Buttons - - - - -	59
3.2.8 The Setup menu - - - - -	59
3.3 Tutorial: Creating and Compiling a C Program - - - - -	60
3.4 The HTLPIC editor - - - - -	65
3.4.1 Frame - - - - -	66
3.4.2 Content Region - - - - -	66
3.4.3 Status Line - - - - -	66
3.4.4 Keyboard Commands - - - - -	67
3.4.5 Block Commands - - - - -	69
3.4.6 Clipboard Editing - - - - -	71
3.4.6.1 Selecting Text - - - - -	71
3.4.6.2 Clipboard Commands - - - - -	72
3.5 HTLPIC menus - - - - -	73
3.5.1 <<>> menu - - - - -	73
3.5.2 File menu - - - - -	73
3.5.3 Edit menu - - - - -	74
3.5.4 Options menu - - - - -	76
3.5.5 Compile menu - - - - -	78
3.5.6 Make menu - - - - -	81
3.5.7 Run menu - - - - -	85
3.5.8 Utility menu - - - - -	85
3.5.9 Help menu - - - - -	88

4 - PICL Command Line Compiler Driver- - - - - 91

4.1 Long Command Lines - - - - -	91
4.2 Default Libraries - - - - -	92
4.3 Standard Run-Time Startoff - - - - -	92
4.4 PICL Compiler Options - - - - -	92
4.4.1 <i>-processor</i> : Define processor - - - - -	92
4.4.2 <i>-AAHEX</i> : Generate American Automation Symbolic Hex - - - - -	92
4.4.3 <i>-ASMLIST</i> : Generate Assembler .LST Files - - - - -	94
4.4.4 <i>-BIN</i> : Generate Binary Output File - - - - -	94
4.4.5 <i>-C</i> : Compile to Object File - - - - -	94
4.4.6 <i>-CRfile</i> : Generate Cross Reference Listing- - - - -	95
4.4.7 <i>-D24</i> : Use 24-bit Doubles - - - - -	95
4.4.8 <i>-D32</i> : Use 32-bit Doubles - - - - -	95
4.4.9 <i>-Dmacro</i> : Define Macro- - - - -	95
4.4.10 <i>-E</i> : Define Format for Compiler Errors - - - - -	96
4.4.10.1 Using the <i>-E</i> Option - - - - -	96
4.4.10.2 Modifying the Standard <i>-E</i> Format - - - - -	96
4.4.10.3 Redirecting Errors to a File - - - - -	97
4.4.11 <i>-Efile</i> : Redirect Compiler Errors to a File - - - - -	97
4.4.12 <i>-FAKELOCAL</i> - - - - -	98
4.4.13 <i>-Gfile</i> : Generate Source Level Symbol File - - - - -	98
4.4.14 <i>-HELP</i> : Display Help - - - - -	98
4.4.15 <i>-Ipath</i> : Include Search Path - - - - -	99
4.4.16 <i>-INTEL</i> : Generate INTEL Hex File - - - - -	99
4.4.17 <i>-Llibrary</i> : Scan Library - - - - -	99
4.4.18 <i>-L-option</i> : Specify Extra Linker Option- - - - -	99
4.4.19 <i>-Mfile</i> : Generate Map File - - - - -	100
4.4.20 <i>-MOT</i> : Generate Motorola S-Record HEX File - - - - -	100
4.4.21 <i>-Nsize</i> : Identifier Length- - - - -	100
4.4.22 <i>-O</i> : Invoke Optimizer - - - - -	100
4.4.23 <i>-Ofile</i> : Specify Output File- - - - -	100
4.4.24 <i>-P</i> : Pre-process Assembly Files - - - - -	100
4.4.25 <i>-PRE</i> : Produce Pre-processed Source Code - - - - -	101
4.4.26 <i>-PROTO</i> : Generate Prototypes - - - - -	101
4.4.27 <i>-PSECTMAP</i> : Display Complete Memory Usage - - - - -	102
4.4.28 <i>-q</i> : Quiet Mode - - - - -	102
4.4.29 <i>-S</i> : Compile to Assembler Code - - - - -	103

4.4.30 -SIGNED_CHAR: Make Char Type Signed - - - - -	103
4.4.31 -STRICT: Strict ANSI Conformance - - - - -	103
4.4.32 -TEK: Generate Tektronix HEX File - - - - -	103
4.4.33 -Umacro: Undefine a Macro - - - - -	103
4.4.34 -UBROF: Generate UBROF Format Output File- - - - -	103
4.4.35 -V: Verbose Compile- - - - -	104
4.4.36 -Wlevel: Set Warning Level - - - - -	104
4.4.37 -X: Strip Local Symbols - - - - -	104
4.4.38 -Zg[level]: Global Optimization - - - - -	104
 5 - Features and Runtime Environment - - - - -	 105
5.1 Divergence from the ANSI C Standard - - - - -	105
5.2 Processor Support - - - - -	105
5.3 Standard Libraries - - - - -	105
5.4 Output File Formats - - - - -	106
5.5 Symbol Files - - - - -	106
5.6 Predefined Macros - - - - -	107
5.7 Configuration Fuses - - - - -	107
5.8 ID Locations - - - - -	108
5.9 Bit Instructions - - - - -	108
5.9.1 The OPTION instruction- - - - -	109
5.9.2 5The TRIS instructions - - - - -	109
5.10 Oscillator calibration constants - - - - -	110
5.11 Supported Data Types - - - - -	110
5.11.1 Radix Specifiers - - - - -	111
5.11.2 Bit Data Types- - - - -	111
5.11.2.1 Using Bit-Addressable Registers - - - - -	112
5.11.3 8-Bit Integer Data Types - - - - -	113
5.11.4 16-Bit Integer Data Types - - - - -	113
5.11.5 32-Bit Integer Data Types - - - - -	113
5.11.6 Floating Point - - - - -	114
5.12 Absolute Variables - - - - -	115
5.13 Structures and Unions - - - - -	115
5.13.1 Structure Qualifiers - - - - -	115
5.13.2 Bit Fields in Structures - - - - -	116
5.14 Strings In ROM and RAM - - - - -	117
5.15 Const and Volatile Type Qualifiers - - - - -	117

5.16 Placement and access of ROM objects - - - - -	-118
5.16.1 Midrange PICs - - - - -	118
5.17 Special Type Qualifiers - - - - -	-118
5.17.1 <i>Persistent</i> Type Qualifier - - - - -	119
5.17.2 <i>Bank1</i> , <i>Bank2</i> and <i>Bank3</i> Type Qualifiers- - - - -	119
5.18 Pointers - - - - -	-119
5.18.1 Midrange Pointers- - - - -	119
5.18.2 Combining Type Qualifiers and Pointers - - - - -	120
5.18.3 Const Pointers - - - - -	121
5.19 Implementation-defined behaviour - - - - -	-121
5.19.1 Shifts applied to integral types - - - - -	121
5.19.2 Division and modulus with integral types - - - - -	121
5.20 Interrupt Handling in C - - - - -	-122
5.20.1 Midrange Interrupt Functions - - - - -	122
5.20.2 Context Saving on Interrupts- - - - -	122
5.20.2.1 MidRange Context Saving - - - - -	-123
5.20.3 Context Retrieval - - - - -	123
5.20.4 Interrupt Levels - - - - -	123
5.21 Mixing C and Assembler Code - - - - -	-125
5.21.1 External Assembly Language Functions - - - - -	125
5.21.2 Accessing C objects from within assembler- - - - -	126
5.21.3 #asm, #endasm and asm() - - - - -	127
5.22 Signature Checking - - - - -	-128
5.23 Linking Programs - - - - -	-129
5.24 Memory Usage - - - - -	-129
5.25 Register Usage - - - - -	-129
5.26 Function Argument Passing - - - - -	-130
5.27 Function Return Values - - - - -	-131
5.27.1 8-Bit Return Values - - - - -	131
5.27.2 16-Bit and 32-bit Return Values - - - - -	131
5.27.3 Structure Return Values - - - - -	132
5.28 Function Calling Convention - - - - -	-132
5.29 Local Variables - - - - -	-133
5.29.1 Auto Variables - - - - -	133
5.29.2 Static Variables - - - - -	134
5.30 Compiler Generated Psects - - - - -	-134
5.31 Runtime Startoff Modules - - - - -	-136
5.31.1 The powerup Routine - - - - -	136

5.32 Linker-Defined Symbols - - - - -	137
5.33 Preprocessor Directives - - - - -	138
5.34 Pragma Directives - - - - -	138
5.34.1 The #pragma jis and nojis Directives - - - - -	138
5.34.2 The #pragma printf_check Directive - - - - -	138
5.34.3 The #pragma psect Directive - - - - -	138
5.34.4 The #pragma regsused Directive - - - - -	141
5.35 Standard I/O Functions and Serial I/O - - - - -	142
5.36 MPLAB-specific Debugging Information - - - - -	142

6 - The PICL Macro Assembler - - - - - 153

6.1 Assembler Usage - - - - -	153
6.2 Assembler Options - - - - -	153
6.3 PICL Assembly Language - - - - -	156
6.3.1 Additional Mnemonics - - - - -	156
6.3.2 Assembler Format Deviations - - - - -	156
6.3.3 Character Set - - - - -	156
6.3.4 Constants - - - - -	156
6.3.4.1 Numeric Constants - - - - -	156
6.3.4.2 Character Constants - - - - -	157
6.3.5 Delimiters - - - - -	157
6.3.6 Special Characters- - - - -	157
6.3.7 Identifiers- - - - -	157
6.3.7.1 Significance of Identifiers - - - - -	157
6.3.7.2 Assembler-Generated Identifiers - - - - -	158
6.3.7.3 Location Counter - - - - -	158
6.3.7.4 Register Symbols - - - - -	158
6.3.7.5 Labels - - - - -	158
6.3.7.6 Symbolic Labels - - - - -	158
6.3.7.7 Numeric Labels - - - - -	159
6.3.8 Strings - - - - -	160
6.3.9 Expressions - - - - -	160
6.3.10 Statement Format - - - - -	160
6.3.11 Program Sections - - - - -	160
6.3.12 Assembler Directives- - - - -	162
6.3.12.1 GLOBAL - - - - -	162
6.3.12.2 END - - - - -	164

6.3.12.3 PSECT	-164
6.3.12.4 ORG	-166
6.3.12.5 EQU	-166
6.3.12.6 SET	-166
6.3.12.7 DEFL	-166
6.3.12.8 DB	-166
6.3.12.9 DW	-167
6.3.12.10 DS	-167
6.3.12.11 FNADDR	-167
6.3.12.12 FNARG	-167
6.3.12.13 FNBREAK	-168
6.3.12.14 FNCALL	-168
6.3.12.15 FNCONF	-168
6.3.12.16 FNINDIR	-168
6.3.12.17 FNSIZE	-169
6.3.12.18 FNROOT	-169
6.3.12.19 IF, ELSIF, ELSE and ENDIF	-169
6.3.12.20 MACRO and ENDM	-170
6.3.12.21 LOCAL	-171
6.3.12.22 ALIGN	-172
6.3.12.23 REPT	-172
6.3.12.24 IRP and IRPC	-172
6.3.12.25 PAGESEL	-174
6.3.12.26 PROCESSOR	-174
6.3.12.27 SIGNAT	-174
6.3.13 Macro Invocations	174
6.3.14 Assembler Controls	174
6.3.14.1 COND	-175
6.3.14.2 EXPAND	-175
6.3.14.3 INCLUDE	-175
6.3.14.4 LIST	-175
6.3.14.5 NOCOND	-176
6.3.14.6 NOEXPAND	-176
6.3.14.7 NOLIST	-176
6.3.14.8 NOXREF	-176
6.3.14.9 PAGE	-176
6.3.14.10 SPACE	-176
6.3.14.11 SUBTITLE	-176

6.3.14.12 TITLE	- - - - -	176
6.3.14.13 XREF	- - - - -	177
7 - Linker and Utilities Reference Manual	- - - - -	179
7.1 Introduction	- - - - -	179
7.2 Relocation and Psects	- - - - -	179
7.3 Program Sections	- - - - -	179
7.4 Local Psects	- - - - -	180
7.5 Global Symbols	- - - - -	180
7.6 Link and load addresses	- - - - -	180
7.7 Operation	- - - - -	181
7.7.1 Numbers in linker options	- - - - -	182
7.7.2 -Aclass=low-high,...	- - - - -	182
7.7.3 -Cx	- - - - -	183
7.7.4 -Cpsect=class	- - - - -	183
7.7.5 -Dclass=delta	- - - - -	183
7.7.6 -Dsymfile	- - - - -	183
7.7.7 -Eerrfile	- - - - -	183
7.7.8 -F	- - - - -	183
7.7.9 -Gspec	- - - - -	184
7.7.10 -Hsymfile	- - - - -	184
7.7.11 -H+symfile	- - - - -	184
7.7.12 -Jerrcount	- - - - -	184
7.7.13 -K	- - - - -	185
7.7.14 -I	- - - - -	185
7.7.15 -L	- - - - -	185
7.7.16 -LM	- - - - -	185
7.7.17 -Mmapfile	- - - - -	185
7.7.18 -N, -Ns and -Nc	- - - - -	185
7.7.19 -Ooutfile	- - - - -	185
7.7.20 -Pspec	- - - - -	185
7.7.21 -Qprocessor	- - - - -	187
7.7.22 -S	- - - - -	187
7.7.23 -Sclass=limit[, bound]	- - - - -	187
7.7.24 -Usymbol	- - - - -	187
7.7.25 -Vavmap	- - - - -	188
7.7.26 -Wnum	- - - - -	188

7.7.27 -X	188
7.7.28 -Z	188
7.8 Invoking the Linker	-188
7.9 Map Files	-189
7.9.1 Call Graph Information	190
7.10 Librarian	-192
7.10.1 The Library Format	192
7.10.2 Using the Librarian	192
7.10.3 Examples	193
7.10.4 Supplying Arguments	194
7.10.5 Listing Format	194
7.10.6 Ordering of Libraries	194
7.10.7 Error Messages	195
7.11 Objtohex	-195
7.11.1 Checksum Specifications	195
7.12 Cref	-196
7.12.1 -Fprefix	197
7.12.2 -Hheading	197
7.12.3 -Llen	197
7.12.4 -Ooutfile	197
7.12.5 -Pwidth	197
7.12.6 -Sstoplist	197
7.12.7 -Xprefix	198
7.13 Memmap	-198
7.13.1 Using MEMMAP	198
7.13.1.1 -P	-198
7.13.1.2 -Wwid	-199
8 - Index	201

Table 2 - 1 - Configuration files	18
Table 2 - 2 - Input file types	20
Table 2 - 3 - clist output	21
Table 2 - 4 - Pre-processor output	22
Table 2 - 5 - Intermediate and Support files	22
Table 2 - 6 - Parser output	23
Table 2 - 7 - Code generator output	24
Table 2 - 8 - Assembler output	26
Table 2 - 9 - Assembler listing	27
Table 2 - 10 - Output formats	29
Table 3 - 1 - Colour values	54
Table 3 - 2 - Colour attributes	54
Table 3 - 3 - Colour coding settings	55
Table 3 - 4 - Menu system key and mouse actions	55
Table 3 - 5 - HTLPIC menu hot keys	57
Table 3 - 6 - Resize mode keys	58
Table 3 - 7 - Editor keys	68
Table 3 - 8 - Block operation keys	69
Table 3 - 9 - Macros usable in user commands	88
Table 4 - 1 - PICL File Types	91
Table 4 - 2 - PICL Options	93
Table 4 - 3 - Processors	94
Table 4 - 4 - Error Format Specifiers	96
Table 5 - 1 - Standard Libraries	106
Table 5 - 2 - Output File Formats	107
Table 5 - 3 - Predefined CPP Symbols	108
Table 5 - 4 - Data Types	111
Table 5 - 5 - Radix Formats	111
Table 5 - 6 - Floating Point Formats	114
Table 5 - 7 - IEEE 754 32-bit and 24-bit Examples	114
Table 5 - 8 - Integral division	122
Table 5 - 9 - Standard Run-time Startoff Modules	136
Table 5 - 10 - Preprocessor directives	139

Table 5 - 11 - Pragma Directives	140
Table 5 - 12 - Valid regsused Register Names	142
Table 5 - 13 - Supported STDIO Functions.	142
Table 6 - 1 - ASPIC Assembler options	154
Table 6 - 2 - ASPIC Numbers and bases	156
Table 6 - 3 - Operators.	161
Table 6 - 4 - ASPIC Statement formats	161
Table 6 - 5 - ASPIC Directives (pseudo-ops)	163
Table 6 - 6 - PSECT flags	164
Table 6 - 7 - ASPIC Assembler controls	175
Table 6 - 8 - LIST Control Options	176
Table 7 - 1 - Linker Options	181
Table 7 - 2 - Librarian Options	193
Table 7 - 3 - Librarian Key Letter Commands	193
Table 7 - 4 - Objtohex Options	196
Table 7 - 5 - Cref Options	197
Table 7 - 6 - Memmap options	198

Figure 2 - 1 - Compilation overview	19
Figure 3 - 1 - HTLPIC Startup Screen.	51
Figure 3 - 2 - Setup Dialogue	60
Figure 3 - 3 - LED Flashing program in HTLPIC.	62
Figure 3 - 4 - HTLPIC File Menu	63
Figure 3 - 5 - Error window.	64
Figure 3 - 6 - HTLPIC Edit Menu.	75
Figure 3 - 7 - Options Menu	77
Figure 3 - 8 - HTLPIC Compile Menu	79
Figure 3 - 9 - HTLPIC Make Menu.	82
Figure 3 - 10 - HTLPIC Run Menu.	86
Figure 3 - 11 - HTLPIC Utility Menu	87
Figure 3 - 12 - HTLPIC Help Menu	89
Figure 5 - 1 - PIC Standard Library Naming Convention.	106

Introduction

1.1 Compiler Limitations

This is the PIC C manual altered for use with the Lite version of the PIC Compiler. The PIC Lite was designed primarily for the use of students and hobbyists who needed an inexpensive PIC compiler, for educational and small projects. The compiler itself is limited to only the 16C84, 16F84, 16F84A and the 16F627 Microchip processors, it also does not come with any library source code. You can perform 24 or 32 bit float calculations, though there is no support for printing (printf) *long* or *float* numbers.

1.2 Typographic conventions

Throughout this manual computer prompts, responses, file names and code will be printed in constant spaced type. Particularly useful points and new terms will be emphasised using *italicised type*. With a window based program like HPD, some concepts are difficult to convey in text. These will be introduced using short tutorials and sample screen displays with references to menu items in **bold**.

1.3 Using This Manual

This manual is a comprehensive guide and reference to using PIC LITE C. The chapters included are as follows:

- ☐ Tutorials to aid in the understanding and usage of HI-TECH's C cross compilers
- ☐ How to use the HI-TECH Professional Development (HPD) environment
- ☐ How to use the PIC LITE C command-line interface
- ☐ In-depth description of the C compiler
- ☐ How to use the assembler
- ☐ How to use the linker and other utilities

Tutorials

The following are tutorials to aid in the understanding and usage of HI-TECH's C cross compilers. These tutorials should be read in conjunction with the appropriate sections in the manual as they are aimed at giving a general overview of certain aspects of the compiler. Some of the tutorials here are generic to all HI-TECH C compilers and may include information not specific for the compiler you are using.

2.1 Overview of the compilation process

This tutorial gives an overview of the compilation process that takes place with HI-TECH C compilers in terms of how the input source files are processed. The origin of files that are produced by the compiler is discussed as well as their content and function.

2.1.1 Compilation

When a program is compiled, it is done so by many separate applications whose operations are controlled by either the *command-line driver* (CLD) or *HPD driver*¹ (HPD). In either case, HPD or the CLD take the options specified by the programmer (menu options in the case of HPD, or command-line arguments for the CLD) to determine which of the internal applications need to be executed and what options should be sent to each. When the term *compiler* is used, this is intended to denote the entire collection of applications and driver that are involved in the process. In the same way, *compilation* refers to the complete transformation from input to output by the compiler. Each application and its function is discussed further on in this document.

The compiler drivers use several files to store options and information used in the compilation process and these file types are shown in Table 2 - 1 on page 18. The HPD driver stores the compiler options into a project file which has a ".prj" extension. HPD itself stores its own configurational settings in an INI file, e.g. `HPD51.ini` in the `bin` directory. This file stores information such as colour values and mouse settings. Users who wish to use the CLD can store the command line arguments in a DOS batch file.

Some compilers come with chip info files which describe the memory arrangements of different chip types. If necessary this file can be edited to create new chip types which can then be selected with the appropriate command-line option of from the **select processor...** menu. This file will also have a ".ini" extension and is usually in the `lib` directory.

1. The command line driver and HPD driver have processor-specific names, such as `picc`, `c51`, or `HPDXA`, `HPDPIC` etc.

The compilation process is discussed in the following sections both in terms of what takes place at each

Table 2 - 1 Configuration files

extension	name	contents
.prj	project file	compiler options stored by HPD driver
.ini	HPD initialisation file	HPD environment settings
.bat	batch file	command line driver options stored as DOS batch file
.ini	chip info file	information regarding chip families

stage and the files that are involved. Reference should be made to Figure 2 - 1 on page 19 which shows the block diagram of the internal stages of the HI-TECH compiler, and the tables of file types throughout this tutorial which list the filename extension² used by different file formats and the information which the file contains. Note that some older HI-TECH compilers do not include all the applications discussed below.

The internal applications generate output files and pass these to the next application as indicated in the figure. The arrows from one application (drawn as ellipses) to another is done via temporary files that have non-descriptive names such as \$\$_003361.001. These files are temporarily stored in a directory pointed to by the DOS environment variable TEMP. Such a variable is created by a set DOS command. These files are automatically deleted by the driver after compilation has been completed.

2.1.2 The compiler input

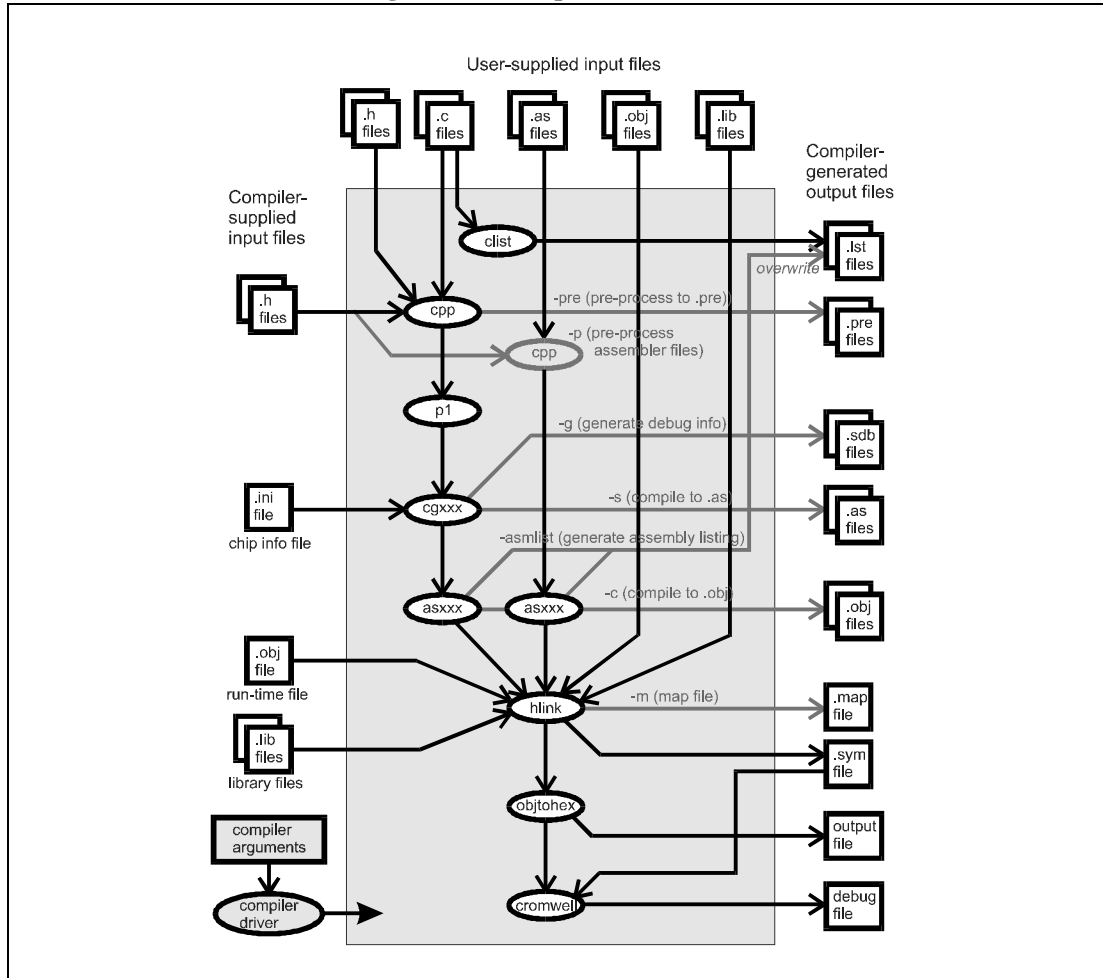
The user supplies several things to the compiler to make a program: the input files and the compiler options, whether using the CLD or HPD. The compiler accepts many different input file types. These are discussed below.

It is possible, and indeed in a large number of projects, that the only files supplied by the user are *C source files* and possibly accompanying header files. It is assumed that anyone using our compiler is familiar with the syntax of the C language. If not, there is a seemingly endless selection of texts which cover this topic. C source files used by the HI-TECH compiler must use the extension ".c" as this extension is used by the driver to determine the file's type. C source files can be listed in any order on the command line if using the CLD, or entered into the **source file list...** dialogue box if using HPD.

A *header file* is usually a file which contains information related to the program, but which will not directly produce executable code when compiled. Typically they include declarations (as opposed to definitions) for functions and data types. These files are included into C source code by a pre-processor

2. The extensions listed in these tables are in lower case. DOS compilers do not distinguish between upper- and lower-case file names and extensions, but in the interest of writing portable programs you should use lower-case extensions in file names and in references to these files in your code as UNIX compilers do handle case correctly.

Figure 2 - 1 Compilation overview



directive and are often called *include files*. Since header files are referenced by a command that includes the file's name and extension (and possibly a path), there are no restrictions as to what this name can be although convention dictates a ".h" extension.

Although executable C code may be included into a source file, a file using the extension ".h" is assumed to have non-executable content. Any C source files that are to be included into other source files should still retain a ".c" extension. In any case, the practise of including one source file into another is best

avoided as it makes structuring the code difficult, and it defeats many of the advantages of having a compiler capable of handling multiple-source files in the first place. Header files can also be included into assembler files. Again, it is recommended that the files should only contain assembler declarations.

Table 2 - 2 Input file types

extension	name	content
.c	C source file	C source conforming to the ANSI standard possibly with extensions allowed by HI-TECH C
.h	header file	C/assembler declarations
.as	assembler file	assembler source conforming to the HI-TECH assembler format
.obj	(relocatable) object file	pre-compiled C or assembler source as HI-TECH relocatable object file
.lib	library file	pre-compiled C or assembler source in HI-TECH library format

HI-TECH compilers comes with many header files which are stored in a separate directory of the distribution. Typically user-written header files are placed in the directory that contains the sources for the program. Alternatively they can be placed into a directory which can be searched by using a `-I` (**CPP include paths...**) option.

An *assembler file* contains assembler *mnemonics* which are specific to the processor for which the program is being compiled. Assembler files may be derived from C source files that have been previously compiled to assembler, or may be hand-written and highly-prized works of art that the programmer has developed. In either case, these files must conform to the format expected of the HI-TECH assembler that is part of the compiler. This processor-dependence makes assembly files quite unportable and they should be avoided if C source can be made to perform the task at hand. Assembler files must have a ".as" extension as this is used by the compiler driver to determine the file's type. Assembler files can be listed in any order on the command line if using the CLD, or entered into the **source file list...** dialogue box if using HPD, along with the C source files.

The compiler drivers can also be passed pre-compiled HI-TECH object files as input. These files are discussed below in Section 2.1.2.1 on page 21. These files must have a ".obj" extension. Object files can be listed in any order on the command line if using the CLD, or entered into the **object file list...** dialogue box if using HPD. You should not enter the names of object files here that have been compiled from source files already in the project, only include object files that have been pre-compiled and have no corresponding source in the project, such as the run-time file should be listed.

Commonly used program routines can be compiled into a file called a *library file*. These files are more convenient to handle and can be accessed quickly by the compiler. The compiler can accept library files directly like other source files. A ".lib" extension indicates the type of the file and so library files must

be named in this way. Library files can be listed in any order on the command line if using the CLD, or entered into the **library file list...** dialogue box if using HPD.

The HI-TECH library functions come pre-compiled in a library format and are stored in a special directory in your distribution.

2.1.2.1 Steps before linking

Of all the different types of files that can be accepted by the compiler, it is the C source files that require the most processing. The steps involved in compiling the C source files are examined first.

For each C source file, a *C listing file* is produced by an application called `clist`. The listing files contain the C source lines preceded by a line number before any processing has occurred. The C listing for a small test program called `main.c` is shown in Table 2 - 3 on page 21.

Table 2 - 3 clist output

C source	C listing
<code>#define VAL 2</code>	<code>1: #define VAL 2</code>
<code>int a, b = 1;</code>	<code>2:</code>
	<code>3: int a, b = 1;</code>
<code>void</code>	<code>4:</code>
<code>main(void)</code>	<code>5: void</code>
<code>{</code>	<code>6: main(void)</code>
<code>/* set starting value */</code>	<code>7: {</code>
<code>a = b + VAL;</code>	<code>8: /* set starting value */</code>
<code>}</code>	<code>9: a = b + 2;</code>
	<code>10: }</code>

The input C source files are also passed to the *preprocessor*, `cpp`. This application has the job of preparing the C source for subsequent interpretation. The tasks performed by `cpp` include removing comments and multiple spaces (such as tabs used in indentation) from the source, and executing any pre-processor directives in the source. Directives may, for example, replace macros with their replacement text or remove source if certain conditions are not true. The pre-processor also copies header files, whether user- or compiler-supplied, into the source. Table 2 - 4 on page 22 shows pre-processor output for the test program.

The output of the pre-processor is C source, but it may contain code which has been included by the pre-processor from other files and only contain code if the pre-processor evaluates specific conditions to be true. The pre-processor output is often referred to as a *module* or *translational unit*. The term "module" is sometimes used to describe the actual source file from which the "true" module is created. This is not strictly correct, but the meaning is clear enough.

The code generation that follows operate on the `cpp` output module, not the C source and so special steps must be taken to be able to reconcile errors and their position in the original C source files. The `# 1 main.c` line in the pre-processor output is included by the pre-processor to indicate the file name

and line number in the C source file that corresponds to this position. Notice in this example that the comment and macro definition have been removed, but blank lines take their place so that line numbering information is kept intact.

Like all compiler applications, the pre-processor is controlled by the compiler driver (either the CLD or

2

Table 2 - 4 Pre-processor output

C source	Pre-processed output
<pre>#define VAL 2 int a, b = 1; void main(void) { /* set starting value */ a = b + VAL; }</pre>	<pre># 1 "main.c" int a, b = 1; void main(void) { a = b + 2; }</pre>

HPD). The type of information that the driver supplies the pre-processor includes directories to search for header files that are included into the source file, and the size of basic C objects (such as `int`, `double`, `char *`, etc.) using the `-S`, `-SP` options so that the pre-processor can evaluate pre-processor directives which contain a `sizeof (type)` expression. The output of the pre-processor is not normally seen unless the user uses the `-pre` option in which case the compiler output can then be re-directed to file.

The output of `cpp` is passed to `p1`, the *parser*. The parser starts the first of the hard work involved with

Table 2 - 5 Intermediate and Support files

extension	name	contents
.pre	pre-processed file	C source or assembler after the pre-processing stage
.lst	C listing file	C source with line numbers
.lst	assembler listing	C source with corresponding assembler instructions
.map	map file	symbol and psect relocation information generated by the linker
.err	error file	compiler warnings and errors resulting from compilation
.rlf	relocation listing file	information necessary to update list file with absolute addresses
.sdb	symbolic debug file	object names and types for module
.sym	symbol file	absolute address of program symbols

turning the description of a program written in the C language into the actual executable itself consisting

of assembler instructions. The parser scans the C source code to ensure that it is valid and then replaces C expressions with a modified form of these. (The description of code generation that follows need not be followed to understand how to use the HI-TECH compiler, but has been included for curious readers.)

For example the expression `a = b + 2` is re-arranged to a prefix notation like `= a + b 2`. This notation can easily be interpreted as a tree with `=` at the apex, `a` and `+` being branches below this, and `b` and `2` being sub-branches of the addition. The output of the parser is shown in Table 2 - 6 on page 23 for our small C program. The assignment statement in the C source has been highlighted as well as the output the parser generates for this statement. Notice that already, the global symbols in the parser output have had an underscore character pre-pended to their name. From now on, reference will be made to them using these symbols. The other symbols in this highlighted line relate to the constant. The ANSI standard states that the constant `2` in the source should be interpreted as a signed `int`. The parser ensures this is the case by casting the constant value. The `->` symbol represents the cast and the `'i` represents the type. Line numbering, variable declarations and the start and end of a function definition can be seen in this output.

It is the parser that is responsible for finding a large portion of the errors in the source code. These errors

Table 2 - 6 Parser output

C source	Parsed output
<pre>#define VAL 2 int a, b = 1; void main(void) { /* set starting value */ a = b + VAL; }</pre>	<pre>Version 3.2 HI-TECH Softwa... "3 main.c [v _a `i 1 e] [v _b `i 1 e] [i _b -> I `i] "7 [v _main `(v 1 e] { [e :U _main] [f] "9 [; ;main.c: 9: b = a + 2; [e = _a + _b -> 2 `i] "10 [; ;main.c: 10: } [e :UE 1] }</pre>

will relate to the syntax of the source code. The parser also reports warnings if the code is unusual.

The parser passes its output directly to the next stage in the compilation process. There are no driver options to force the parser to generate parsed-source output files as these files contain no useful information for the programmer.

Now the tricky part of the compilation: code generation. The *code generator* converts the parser output into assembler mnemonics. This is the first step of the compilation process which is processor-specific. Whereas all HI-TECH pre-processors and parsers have the same name and are in fact the same application, the code generators will have a specific, processor-based name, for example *cgpic*, or *cg51*.

The code generator uses a set of rules, or *productions*, to produce the assembler output. To understand

Table 2 - 7 Code generator output

C source	assembler (XA) code
<pre>#define VAL 2 int a, b = 1; void main(void) { /* set starting value */ a = b + VAL; }</pre>	<pre>psect text _main: ;main.c: 9: a = b + 2; global _b mov r0,#_b movc.w r1,[r0+] adds.w r1,#02h mov.w _a,r1</pre>

how a production works, consider the following analogy of a production used to generate the code for the addition expression in our test program. "If you can get one operand into a register" and "one operand is a int constant" then here is the code that will perform a 2-byte addition of them. Here, each quoted string would represent a sub-production which would have to be matched. The first string would try to get the contents of `_a` into a register by matching further sub-productions. If it cannot, this production cannot be used and another will be tried. If all the sub-productions can be met, then the code that they produce can be put together in the order specified by the production tree. Not all productions actually produce code, but are necessary for the matching process.

If no matching production/subproductions can be found, the code generator will produce a "Can't generate code for this expression" error. This means that the original C source code was legal and that the code generator did try to produce assembler code for it, but that in this context, there are no productions which can match the expression.

Typically there may be around 800 productions to implement a full code generator. There were about a dozen matching productions to generate code for the statement highlighted in Table 2 - 7 on page 24 for the XA code generator. It checked about 70 productions which were possible matches before finding a solution. The exact code generation process is too complex to describe in this document and is not required to be able to use the compiler efficiently.

The user can stop the compilation process after code generation by issuing a `-s (compile to .as)` option to the driver. In this case, the code generator will leave behind assembler files with a ".as" extension.

Table 2 - 7 on page 24 shows output generated by the XA code generator. Only the assembler code for the opening brace of `_main` and the highlighted source line is shown. This output will be different for other compilers and compiler options.

The code generator may also produce debugging information in the form of an ".sdb" file. This operation is enabled by using the `-g` (**source level debug info**) option. One debug file is produced for each module that is being compiled. These ASCII files contain information regarding the symbols defined in each module and can be used by debugging programs. Table 2 - 5 on page 22 shows the debug files that can be produced by the compiler at different stages of the compilation. Several of the output formats also contain debugging information in addition to the code and data.

The code generator optionally performs one other task: optimization. HI-TECH compilers come with several different optimizer stages. The code generator is responsible for *global optimization* which can be enabled using a `-Zg` (**global optimization**) option. This optimization is performed on the parsed source. Amongst other things, this optimization stage allocates variables to registers whenever possible and looks for constants that are used consecutively in source code to avoid reloading these values unnecessarily.

Assembly files are the first files in the compilation process that make reference to *psects*, or program sections. The code generator will generate the psect directives in which code and data will be positioned.

The output of the code generator is then passed to the *assembler* which converts the ASCII representation of the processor instructions - the ASCII mnemonics - to binary *machine code*. The assembler is specific for each compiler as has a processor-dependent name such as `aspc` or `asxa`. Assembler code also contains assembler directives which will be executed by the assembler. Some of these directives are to define ROM-based constants, others define psects and others declare global symbols.

The assembler is optionally preceded by an optimization of the generated assembler. This is the peephole optimization. With some HI-TECH compilers the peephole optimizer is contained in the assembler itself, e.g. the PIC assembler, however others have a separate optimization application which is run before the assembler is executed, e.g. `opt51`. Peephole optimization is carried out separately over the assembler code derived from each single function.

In addition to the peephole optimizer, the assembler itself may include a separate assembler optimizer step which attempts to replace long branches with short branches where possible. The `-O` option enables both assembler optimizers, even if they are performed by separate applications, however HPD includes menu items for both optimizer stages (**Peephole optimization** and **Assembler optimization**). If the peephole optimizer is part of the assembler, the assembler optimization item in HPD has no effect.

The output of the assembler is an object file. An *object file* is a formatted binary file which contains machine code, data and other information relating to the module from which it has been generated. Object files come in two basic types: *relocatable* and *absolute* object files. Although both contain

Table 2 - 8 Assembler output

C source	Relocatable object file
#define VAL 2	11 TEXT 22 text 0 13
int a, b;	99 08 00 00 88 10 A9 12 8E 00 00 D6 80
void	12 RELOC 63
main(void)	2 RPSECT data 2
{	9 COMPLEX 0
/* set start...	Key: direct
a = b + VAL;	0x7>=(high bss)
}	9 COMPLEX 1
	((high bss)&0x7)+0x8
	10 COMPLEX 1
	low bss

machine code in binary form, relocatable object files have not had their addresses resolved to be absolute values. The binary machine code is stored as a block for each psect. Any addresses in this area are temporarily stored as 00h. Separate relocation information in the object file indicates where these unresolved addresses lie in the psect and what they represent. Object files also contain information regarding any psects that are defined within so that the linker may position these correctly.

Object files produced by the assembler follow a format which is standard for all HI-TECH compilers, but obviously their contents are machine specific. Table 2 - 8 on page 26 shows several sections of the HI-TECH format relocatable object file that has been converted to ASCII for presentation using the DUMP executable which comes with the compiler. The highlighted source line is represented by the highlighted machine code in the object file. This code is positioned in a psect called `text`. The underlined bytes in the object file are addresses that as yet are unknown and have been replaced with zeros. The lines after the `text` psect in the object file show the information used to resolve the addresses needed by the linker. The two bytes starting at offset 2 and the two single bytes at offset 9 and 10 are represented here and as can be seen, their address will be contained at an address derived from the position of the data and bss psects, respectively.

If a `-asmlist` (**generate assemble listing**) option was specified, the assembler will generate an assembler listing file which contains both the original C source lines and the assembler code that was generated for each line. The assembler listing output is shown in Table 2 - 9 on page 27. Unresolved addresses are listed as being zero with unresolved-address markers `' '` and `"*"` used to indicate that the values are not absolute. Note that code is placed starting from address zero in the new `text` psect. The entire psect will be relocated by the linker.

Some HI-TECH assemblers also generate a *relocatable listing file* (extension: `".rlf"`).³ This contains address information which can be read by the linker and used to update the assembler listing file, if such

3. The generation of this file is not shown in Figure 2 - 1 on page 19 in the interests of clarity.

Table 2 - 9 Assembler listing

C source	Assembler listing
#define VAL 2	10 0000' psect text
int a, b;	11 0000' _main:
void	12 ;main.c: 9: a = b + 2;
main(void)	13 0000' 99 08 0000' mov.w r0,# b
{	14 0004' 88 10 movc.w r1,[r0+]
/* set start...	15 0006' A9 12 adds.w r1,#2
a = b + VAL;	16 0008' 8E 00* 00* mov.w _a,r1
}	17 ;main.c: 10: }
	18 000B' D6 80 ret

a file was created. After linking, the assembler listing file will have addresses and unresolved address markers removed and replaced with absolute addresses.

The above series of steps: pre-processing, parsing, code generation and assembly, are carried out for each C source file passed to the driver in turn. Errors in the code are reported as they are detected. If a file cannot be compiled due to an error, the driver halts compilation of that module after the application that generated the error completes and continues with the next file which were passed to it, starting again with the `clist` application.

For any assembler files passed to the driver, these do not require as much processing as C source files, but they must be assembled. The compiler driver will pass any ".as" files straight to the assembler. If the user specifies the `-p` (**pre-process assembler files**) the assembler files are first run through the C pre-processor allowing the using of all pre-processor directives within assembly code. The output of the pre-processor is then passed to the assembler.

Object and library files passed to the compiler are already compiled and are not processed at all by the first stages of the compiler. They are not used until the link stage which is explained below.

If you are using HPD, *dependency information* can be saved regarding each source and header file by clicking the **save dependency information** switch. When enabled, the HPD driver determines only which files in the project need be re-compiled from the modification dates of the input source files. If the source file has not been changed, the existing object file is used.

2.1.2.2 The link stage

The format of object files are again processor-independent so the linker and other applications discussed below are common across the whole range of HI-TECH compilers. The linker's name is `hlink`.⁴

The tasks of the linker are many. The linker is responsible for combining all the object and library files into a single file. The files operated on by the linker include all the object files compiled from the input

4. Early HI-TECH linkers were called `link`.

C source files and assembler files, plus any object files or library files passed to the compiler driver, plus any run-time object files and library files that the driver supplies. The linker also performs *grouping* and *relocation* of the psects contained in all of the files passed to it, using a relatively complex set of linker options. The linker also resolves symbol names to be absolute addresses after relocation has made it possible to determine where objects are to be stored in ROM or RAM. The linker then adjusts references to these symbols - a process known as address *fixup*. If the symbol address turns out to be too large to fit into the space in the instruction generated by the code generator, a "fixup overflow" error occurs.

The linker can also generate a map file which has detailed information regarding the position of the psects and the addresses assigned to symbols. The linker may also produce a symbol file. These files have a ".sym" extension and are generated when the `-g` (**source level debug info**) option is used. This symbol file is ASCII-based and contains information for the entire program. Addresses are absolute as this file is generated after the link stage.

The output of linkers for compilers which compile for an operating-system based computer are true executable object files that can be loaded and run. The compilation process, however, is not quite finished for a cross compiler. Although the object file produced by `hlink` contains all the information necessary to run the program, the program has to be somehow transferred from the host computer to the embedded hardware.

There are a number of standard formats that have been created for such a task. Emulators and chip programmers often can accept a number of these formats. The Motorola HEX (S record) or Intel HEX formats are common formats. These are ASCII formats allowing easy viewing by any text editor. They include *checksum* information which can be used by the program which downloads the file to ensure that it was transmitted without error. These formats include address information which allows those areas which do not contain data to be omitted from the file. This can make these files significantly smaller than, for example, a binary file.

The `objtohex` application is responsible for producing the output file requested by the user. It takes the absolute object file produced by the linker and produces an output under the direction of the compiler driver. The `objtohex` application can produce a variety of different formats to satisfy most development systems. The output types available with most HI-TECH compilers are shown in Table 2 - 10 on page 29.

In some circumstances, more than one output file is required. In this case an application called `cromwell`, the reformatter, is executed to produce further output files. For example it is commonly used with the PIC compiler to read in the HEX file and the SYM file and produce a COD file.

Table 2 - 10 Output formats

extension	name	content
.hex	Motorola hex	code in ASCII, Motorola S19 record format
.hex	Intel hex	code in ASCII, Intel format
.hex	Tektronix hex	code in ASCII Tek format
.hex	American Auto- mation hex	code and symbol information in binary, American Automa- tion format
.bin	binary file	code in binary format
.cod	Bytecraft COD file	code and symbol information in binary Bytecraft format
.cof	COFF file	code and symbol information in binary common object file format
.ubr	UBROF file	code and symbol information in universal binary relocatable object format
.omf	OMF-51 file	code and symbol information in Intel Object Module For- mat for 8051
.omf	enhanced OMF- 51 file	code and symbol information in Keil Object Module Format for 8051

2.2 Psects and the linker

This tutorial explains how the compiler breaks up the code and data objects in a C program into different parts and then how the linker is instructed to position these into the ROM and RAM on the target.

2.2.1 Psects

As the code generator progresses it generates an assembler file for each C source file that is compiled. The contents of these assembly files include different sections: some containing assembler instructions that represent the C source; others contain assembler directives that reserve space for variables in RAM; others containing ROM-based constants that have been defined in the C source; and others which hold data for special objects such as non-volatile variables, interrupt vectors and configuration words used by the processor. Since there can be more than one input source file there will be similar sections of assembler spread over multiple assembler files which need to be grouped together after all the code generation is complete.

These different sections of assembler need to be grouped in special ways: It makes sense to have all the initialised data values together in contiguous blocks so they can be copied to RAM in one block move rather than having them scattered in-between sections of code; the same applies to uninitialised global objects which have to be allocated a space which is then cleared before the program starts; some code

or objects have to be positioned in certain areas of memory to conform to requirements in the processor's addressing capability; and at times the user needs to be able to position code or data at specific absolute addresses to meet special software requirements. The code generator must therefore include information which indicates how the different assembler sections should be handled and positioned by the linker later in the compilation process.

The method used by the HI-TECH compiler to group and position different parts of a program is to place all assembler instructions and directives into individual, relocatable sections. These sections of a program are known as *psects* - short for program sections. The linker is then passed a series of options which indicate the memory that is available on the target system and how all the psects in the program should be positioned in this memory space.

2.2.1.1 The psect directive

The psect assembler directives (generated by the code generator or manually included in other assembly files) define a new psect. The general form of this directive is shown below.

```
psect name,option,option...
```

It consists of the token `psect` followed by the name by which this psect shall be referred. The name can be any valid assembler identifier and does not have to be unique. That is, you may have several psects with the same name, even in the same file. As will be discussed presently, psects with the same name are usually grouped together by the linker.

The directive options are described in the assembler section of the manual, but several of these will be discussed in this tutorial. The options are instructions to the linker which describe how the psect should be grouped and relocated in the final absolute object file.

Psects which all have the same name imply that their content is similar and that they should be grouped and linked together in the same way. This allows you to place objects together in memory even if they are defined in different files.

After a psect has been defined, the options may be omitted in subsequent psect directives in the same module that use the same name. The following example shows two psects being defined and filled with code and data.

```
psect text,global
begin:
    mov    r0,#10
    mov    r2,r4
    add    r2,#8
psect data
input:
    ds     8
```

```

psect text
next:
    mov  r4,r2
    rrc  r4

```

In this example, the `psect text` is defined including an option to say that this is a global psect. Three assembler instructions are placed into this psect. Another psect is created: `data`. This psect reserves 8 bytes of storage space for data in RAM. The last psect directive will continue adding to the first psect. The options were omitted from the `psect` directive in this example as there has already been a psect directive in this file that defines the options for a psect of this name. The above example will generate two psects. Other assembler files in the program may also create psects which have the same name as those here. These will be grouped with the above by the linker in accordance with the `psect` directive flags.

2.2.1.2 Psect types

Psects come in three broad types: those that will reside permanently in ROM⁵; those that will be allocated space in RAM after the program starts; and those that will reside in ROM, but which will be copied into another reserved space in RAM after the program starts. A combination of code - known as the *run-time* (or *startup*) code - and psect and linker options allow all this to happen.

Typically, psects placed into ROM contain instructions and constant data that cannot be modified. Those psects allocated space in RAM only are for global data objects that do not have to assume any non-zero value when the program starts, i.e. they are uninitialised. Those psects that have both a ROM image and space reserved in RAM are for modifiable, global data objects which are initialised, that is they contain some specific value when the program begins, but that value can be changed by the program during its execution.

The following C source shows two objects being defined. The object `input` will be placed into a data psect; the value 22 will reside in ROM and be copied to the RAM space allocated for `input` by the run-time code. The object `output` will not contribute directly to the ROM image. An area of memory will be reserved for it in RAM and this area will be cleared by the run-time code (`output` will be assigned the value 0).

```

int input = 22;  // an initialised object
int output;     // an uninitialised object

```

Snippets from the assembler listing file show how the 8051XA compiler handles these two objects. Other compilers may produce differently structured code. The psect directive flags are discussed

5. The term "ROM" will be used to refer to any non-volatile memory.

presently, but note that for the initialised object, `input`, the code generator used a `dw` (define word) directive which placed the two bytes of the `int` value (16 and 00) into the output which is destined for the ROM. Two bytes of storage were reserved using the `ds` assembler directive for the uninitialised object, `output`, and no values appear in the output.

```

1      0000'                psect data,class=CODE,space=0,align=0
2                                global _input
3                                align.w
4      0000'                _input:
5      0000' 16 00          dw 22

13     0000'                psect bss,class=DATA,space=1,align=0
14                                global _output
15                                align.w
16     0000'                _output:
17     0000'                ds 2

```

Auto variables and function parameters are local to the function in which they are defined and so are handled different by the compiler. They may be allocated space dynamically (for example on the stack) in which case they are not stored in psects by the compiler.

Two addresses are used to refer to the location of a psect: the *link address* and the *load address*. The link address is the address at which the psect (and any objects or labels within the psect) can be accessed whilst the program is executing. The load address is the address at which the psect will reside in the output file that creates the ROM image, or, alternatively, the address of where the psect can be accessed in ROM.

For the psect types that reside in ROM their link and load address are the same as they reside in ROM and are never copied to a new location. Psects that are allocated space in RAM only will have a link address, but a load address is not applicable. The compiler often makes the load address of these psects the same as the link address. Since no ROM image of these psects is formed, the load address is meaningless and can be ignored. Any access to objects defined in these psects is performed using the link address. The psects that reside in ROM, but are copied to RAM have link and load addresses that are usually different. Any references to symbols or labels in these psects are always made using the link addresses.

2.3 Linking the psects

After the code generator and assembler⁶ have finished their jobs, the object files passed to the linker can be considered to be a mixture of psects that have to be grouped and positioned in the available ROM and

RAM. The linker options indicate the memory that is available and the flags associated with a psect directive indicate how the psects are to be handled.

2.3.1 Grouping psects

There are two psect flags that affect the grouping, or merging, of the psects. These are the `LOCAL` and `GLOBAL` flags. `GLOBAL` is the default and tells the linker that the psects should be grouped together with other psects of the same name to form a single psect. `LOCAL` psects are not grouped in this way unless they are contained in the same module. Two local psects which have the same name, but which are defined in different modules are treated and positioned as separate psects.

2.3.2 Positioning psects

Several psect flags affect how the psects are positioned in memory. Psects which have the same name can be positioned in one of two ways: they can be overlaid one another, or they can be placed so that each takes up a separate area of memory.

Psects which are to be overlaid will use the `OVLRD` psect directive flag. At first it may seem unusual to have overlaid psects as they might destroy other psects' contents as they are positioned, however there are instances where this is desirable.

One case where overlaid psect are used is when the compiler has to use temporary variables. When the compiler has to pass several data objects to, say, a floating-point routine, the floats may need to be stored in temporary variables which are stored in RAM. It is undesirable to have the space reserved if it is not going to be used, so the routines that use the temporary objects are also responsible for defining the area and reserving the space in which these will reside. However several routines may called and hence several temporary areas created. To get around this problem, the psects which contain the directives to reserve space for the objects are defined as being overlaid so that if more than one is defined, they since simply overlap each other.

Another situation where overlaid psects are used is when defining the interrupt vectors. The run-time code usually defines the reset vector, but other vectors are left up to the programmer to initialize. Interrupt vectors are placed into a separate psect (often called `vectors`). Each vector is placed at an offset from the beginning of the vectors area appropriate for the target processor. The offset is achieved via an `ORG` assembler directive which moves the location counter relative to the beginning of the current psect. The macros contained in the header file `<intrpt.h>`, which allow the programmer to define additional interrupt vectors, also place the vectors they define into a psect with this same name, but with different offsets, depended on the interrupt vector being defined. All these psects are grouped and overlaid such that the vectors are correctly positioned from the same address - the start of the vectors psect. This merged psect is then positioned by the linker so that it begins at the start of the vectors area.

-
6. The assembler does not modify psect directives in any way other than encoding the details of each into the object file.

Most other compiler-generated psects are not overlaid and so they will each occupy their own unique address space. Typically these psects are placed one after the other in memory, however there are several psect flags that can alter the positioning of the psects. Some of these psect flags are discussed below.

The RELOC flag is used when psects must be aligned on a boundary in memory. This boundary is a multiple of the value specified with the flag. The ABS flag specifies that the psect is absolute and that it should start at address 0h. Remember, however, that if there are several psects which use this flag, then after grouping only one can actually start at address 0h unless the psects are also defined to be overlaid. Thus ABS means that one of the psects with this name will be located at address 0h, the others following in memory subject to any other psect flags used.

2.3.3 Linker options to position psects

The linker is told of the memory setup for a target program by the linker options that are generated by the compiler driver. The user informs the compiler driver about memory using either the `-A` option⁷ with the command line driver (CLD), or via the **ROM & RAM addresses** dialogue box under HPD. Additional linker options indicate how the psects are to be positioned into the available memory.

The linker options are a little confusing at first, but the following example shows how the options could be built up as a program develops, and then discusses some of the specific schemes used by HI-TECH compilers. When compiling using either the CLD or HPD, a full set of default linker options are used, based on either the `-A` option values, or the **ROM & RAM addresses** dialogue values. In most cases the linker options do not need to be modified.

2.3.3.1 Placing psects at an address

Let us assume that the processor in a target system can address 64 kB of memory and that ROM, RAM and peripherals all share this same block of memory. The ROM is placed in the top 16 kB of memory (C000h - FFFFh); RAM is placed at addresses from 0h to FFFh.

Let us also assume that three object files passed to the linker: one a run-time object file; the others compiled from the programmer's C source code. Each object file contains a compiler-generated `text` psect (a psect called `text`): the psect in one file is 100h bytes long; that from other file is 200h bytes long; that from the run-time object file is 50h long. These psects are to be placed in ROM and all have the simple definition generated by the code generator:

```
psect text,class=CODE
```

7. The `-A` option on the PIC compiler serves a different purpose. Most PIC devices only have internal memory and so a memory option is not required by the compiler. High-end PICs may have external memory, this is indicated to the compiler by using a `-ROM` option to the CLD or by the **RAM & ROM addresses...** dialogue box under HPDPIC. The `-A` option is used to shift the entire ROM image, when using highend devices.

The `CLASS` flag is typically used with these types of psects and is considered later in this tutorial. By default, these psects are `GLOBAL`, hence after scanning all the object files passed to it, the linker will group all the `text` psects together so that they are contiguous⁸ and form one larger `text` psect. The following `-p` linker option could be used to position the `text` psect at the bottom of ROM.

```
-p text=0C000h
```

There is only one address specified with this linker option since the psects containing code are not copied from ROM to RAM at any stage and the link and load addresses are the same.

The linker will relocate the grouped `text` psect so that it starts at address `C000h`. The linker will then define two global symbols with names: `__Ltext` and `__Htext` and equate these with the values: `C000h` and `C350h` which are the start and end addresses, respectively, of the `text` psect group.

Now let us assume that the run-time file and one of the programmer's files contains interrupt vectors. These vectors must be positioned at the correct location for this processor. Our fictitious processor expects its vectors to be present between locations `FFC0h` and `FFFFh`. The reset vector takes up two bytes at address `FFFEh` and `FFFFh`, and the remaining locations are for peripheral interrupt vectors. The run-time code usually defines the reset vector using code like the following.

```
global    start
psect     vectors,ovlrd
org       3Eh
dw        start
```

This assembler code creates a new psect which is called `vectors`. This psect uses the overlaid flag (`OVLRD`) which tells the linker that any other psects with this name should be overlaid with this one, not concatenated with it. Since the psect defaults to being `global`, even `vectors` psects in other files will be grouped with this one. The `org` directive tells the assembler to advance `3Eh` locations into this psect. It does *not* tell the assembler to place the object at address `3Eh`. The final destination of the vector is determined by the relocation of the psect performed by the linker later in the compilation process. The assembler directive `dw` is used to actually place a word at this location. The word is the address of the (global) symbol `start`. (`start` or `powerup` are the labels commonly associated with the beginning of the run-time code.)

In one of the user's source files, the macro `ROM_VECTOR` has been used to supply one of the peripheral interrupts at offset `10h` into the vector area. The macro expands to the following in-line assembler code.

8. Some processors may require word alignment gaps between code or data. These gaps can be handled by the compiler, but are not considered here.

```
global    _timer_isr
psect    vectors,ovlrd
org       10h
dw        _timer_isr
```

2

After the first stages of the compilation have been completed, the linker will group together all the vectors psects it finds in all the object files, but they will all start from the same address, i.e. they are all placed one over the other. The final vectors psect group will contain a word at offset 10h and another at offset 3Eh. The space from 0h to offset 0Fh and in-between the two vectors is left untouched by the linker. The linker options required to position this psect would be:

```
-pvectors=0FFC0h
```

The address given with this option is the base address of the vectors area. The org directives used to move within the vectors psects hence were with respect to this base address.

Both the user's files contain constants that are to be positioned into ROM. The code generator generates the following psect directive which defines the psect in which it store the values.

```
psect const
```

The linker will group all these const psects together and they can be simply placed like the text psects. The only problem is: where? At the moment the text psects end at address C34Fh so we could position the const psects immediately after this at address C350h, but if we modify the program, we will have to continually adjust the linker options. Fortunately we can issue a linker option like the following.

```
-ptext=0C000h,const
```

We have not specified an address for the psect const, so it defaults to being the address immediately after the end of the preceding psect listed in the option, i.e. the address immediately after the end of the text psect. Again, the const psect resides in ROM only, so this one address specifies both the link and load addresses.

Now the RAM psects. The user's object files contain uninitialised data objects. The code generator generates bss psects in which are used to hold the values stored by the uninitialised C objects. The area of memory assigned to the bss psect will have to be cleared before main() is executed.

At link time, all bss psects are grouped and concatenated. The psect group is to be positioned at the beginning of RAM. This is easily done via the following option.

```
-pbss=0h
```

The address 0h is the psect's link address. The load address is meaningless, but will default to the link address. The run-time code will clear the area of memory taken up by the bss psect. This code will use

the symbols `__Lbss` and `__Hbss` to determine the starting address and the length of the area that has to be cleared.

Both the user's source files contain initialised objects like the following.

```
int init = 34;
```

The value 34 has to be loaded into the object `init` before the `main()` starts execution. Another of the tasks of the run-time code is to initialise these sorts of objects. This implies that the initial values must be stored in ROM for use by the run-time code. But the object is a variable that can be written to, so it must be present in RAM once the program is running. The run-time code must then copy the initialised values from ROM into RAM just before `main()` begins. The linker will place all the initial values into ROM in exactly the same order as they will appear in RAM so that the run-time code can simply copy the values from ROM to RAM as a single block. The linker has to be told where in ROM these values should reside as it generates the ROM image, but it must also know where in RAM these objects will be copied to so that it can leave enough room for them and resolve the run-time addresses for symbols in this area.

The complete linker options for our program, including the positioning of the `data` psects, might look like:

```
-ptext=0C000h,const
-pvectors=0FFC0h
-pbss=0h,data/const
```

That is, the `data` psect should be positioned after the end of the `bss` psect in RAM. The address after the slash indicates that this psect will be copied from ROM and that its position in ROM should be immediately after the end of the `const` psect. As with other psects, the linker will define symbols `__Ldata` and `__Hdata` for this psect, which are the start and end link addresses, respectively, that will be used by the run-time code to copy the `data` psect group. However with any psects that have different link and load addresses, another symbol is also defined, in this case: `__Bdata`. This is the load address in ROM of the `data` psect.

2.3.3.2 Exceptional cases

The PIC compiler handles the `data` psects in a slightly different manner. It actually defines two separate psects: one for the ROM image of the `data` psects; the other for the copy in RAM. This is because the length of the ROM image is different to the length of the psect in RAM. (The ROM is wider than the RAM and values stored in ROM may be encoded as `retlw` instructions.) The linker options in this case will contain two separate entries for both psects instead of the one psect with different link and load addresses specified. The names of the `data` psects in RAM are similar to `rdata_0`; those in ROM are like `idata_0`. The digit refers to a RAM bank number.

The link and load addresses reported for psects that contain objects of type `bit` have slightly different meaning to ordinary link and load addresses. In the map file, the link address listed is the link address of the psect specified as a bit address. The load address is the link address specified as a byte address. Bit objects cannot be initialised, so separate link and load addresses are not required. The linker knows to handle these psects differently because of the `BIT` psect flag. Bit psects will be reported in the map file as having a *scale* value of 8. This relates to the number of objects that can be positioned in an addressable unit.

2.3.3.3 Psect classes

Now let us assume that more ROM is added to our system since the programmers have been busy and filled the 16 kB currently available. Several peripheral devices were placed in the area from B000h to BFFFh so the additional ROM is added below this from 7000h to AFFFh. Now there are two separate areas of ROM and we can no longer give a single address for the `text` psects.

What we can now do to take advantage of this extra memory is define several ranges of addresses that can be used by ROM-based psects. This can be done by creating a *psect class*. There are several ways that psects can be linked when using classes. Classes are commonly used by HI-TECH C compilers to position the code or `text` psects. Different strategies are employed by different compilers to better suit the processor architecture for which the compilation is taking place. Some of these schemes are discussed below. If you intend to modify the default linker options or generate your own psects, check the linker options and psect directives generated by the code generator for the specific compiler you are using.

A class can be defined using another linker option. For example to use the additional memory added to our system we could define a class using the linker option:

```
-ACODE=7000h-AFFFh,C000h-FFFFh
```

The option is a `-A` immediately followed by the class name and then a comma-separated list of address ranges. Remember this is an option to the linker, not the CLD. The above example defines two address ranges for a class called `CODE`.

Here is how drivers for the 8051, 8051XA and Z80 compilers define the options passed to the linker to handle the code psects. In large model the 8051 psect definitions for psects that contain code are as follows.

```
psect text,class=CODE
```

The `CLASS` psect flag specifies that the psect `text` is a member of the class called `CODE`.

If a single ROM space has been specified by either not using the `-ROM` option with the CLD or by selecting **single ROM** in the **ROM & RAM addresses** dialogue box under HPD, no class is defined and the psects are linked using a `-p` option as we have been doing above. Having the psects within a

class, but not having that class defined is acceptable, provided that there is a `-p` option to explicitly position the psects after they have been grouped. If there is no class defined and no `-p` option a default memory address is used which will almost certainly be inappropriate.

If multiple ROM spaces have been specified by using either the `-ROMranges` option with the CLD, or specifying the address ranges in the **ROM & RAM addresses** (after selecting the **multiple ROMs** button) dialogue box under HPD, a class is defined by the driver using the `-A` linker option similar to that shown above and the `-p` option is omitted from the options passed to the linker.

The PIC compiler does things a little differently as it has to contend with multiple ROM pages that are quite small. The PIC code generator defines the psects which hold code like the following.

```
psect text0,local,class=CODE,delta=2
```

The DELTA value relates to the ROM width and need not be considered here. The psects are placed in the CODE class, but note that they are made local using the LOCAL psect flag. The psects that are generated from C functions each have unique names which proceed: `text0`, `text1`, `text2` etc. Local psects are not grouped across modules, i.e. if there are two modules, each containing a local psect of the same name, they are treated as separate psects. Local psects cannot be positioned using a `-p` linker option as there can be more than one psect with that name. Local psects must be made members of a class, and the class defined using a `-A` linker option. The PIC works in this way to assist with the placement of the code in its ROM pages. This is discussed further in Section 2.3.4 on page 41.

A few general rules apply when using classes: If, for example, you wanted to place a psect that is not already in a class into the memory that a class occupies, you can replace an address or psect name in a linker `-p` option with a class name. For instance, in the generic example discussed above, the `const` psect was placed after the `text` psect in memory. If you would now like this psect to be positioned in the memory assigned to the CODE class the following linker options could be used.

```
-pconst=CODE
-pvectors=0FFC0h
-pbss=0h,data/CODE
-ACODE=7000h-AFFFh,C000h-FFFFh
```

Note also that the `data` psect's load location has been swapped from after the end of the `const` psect to within the memory assigned to the CODE class to illustrate that the load address can be specified using the class name.

Another class definition that is sometimes seen in PIC linker options specifies three addresses for each memory range. Such an option might look something like:

```
-AENTRY=0h-FFh-1FFh
```

The first range specifies the address range in which the psect must start. The psects are allowed to continue past the second address as long as they do not extend past the last address. For the example above, all psects that are in the `ENTRY` class must start at addresses between 0 and FFh. The psects must end before address 1FFh. No psect may be positioned so that its starting address lies between 100h and 1FFh. The linker may, for example, position two psects in this range: the first spanning addresses 0 to 4Fh and the second starting at 50h and finishing at 138h. Such linker options are useful on some PIC processors (typically baseline PICs) for code psects that have to be accessible to instructions that modify the program counter. These instructions can only access the first half of each ROM page.

2.3.3.4 User-defined psects

Let us assume now that the programmer wants to include a special initialised C object that has to be placed at a specific address in memory, i.e. it cannot just be placed into, and linked with, the `data` psect. In a separate source file the programmer places the following code.

```
#pragma psect data=lut
int lookuptable[] = {0, 2, 4, 7, 10, 13, 17, 21, 25};
```

The pragma basically says, from here onwards in this module, anything that would normally go into the `data` psect should be positioned into a new psect called `lut`. Since the array is initialised, it would normally be placed into `data` and so it will be re-directed to the new psect. The psect `lut` will inherit any psect options (defined by the psect directive flags) which applied to `data`.

The array is to be positioned in RAM at address 500h. The `-p` option above could be modified to include this psect as well.

```
-pbss=0h,data/const,lut=500h/
```

(The load address of the `data` psect has been returned to its previous setting.) The addresses for this psect are given as "500h/". The address "500h" specifies the psect's link address. The load address can be anywhere, but it is desirable to concatenate it to existing psects in ROM. If the link address is not followed by a load address at all, then the link and load addresses would be set to be the same, in this case 500h. The "/", which is not followed by an address, tells the linker that the load address should be immediately after the end of the previous psect's load address in the linker options. In this case that is the `data` psect's load address, which in turn was placed after the `const` psect. So, in ROM will be placed the `const`, `data` and `lut` psect, in that order.

Since this is an initialised data psect, it is positioned in ROM and must be copied to the memory reserved for it in RAM. Although different link and load addresses have been specified with the linker option, the programmer will have to supply the code that actually performs the copy from ROM to RAM. (The data psects normally created by the code generator have code already supplied in the run-time file to copy the psects.) The following is C code which could perform the copy.

```

extern unsigned char *_Llut, *_Hlut, *_Blut;
unsigned char *i;

void copy_my_psect(void)
{
    for(i=_Llut; i<_Hlut; i++, _Blut++)
        *i = *_Blut;
}

```

Note that to access the symbols `__Llut` etc. from within C code, the first underscore character is dropped. These symbols hold the addresses of psects, so they are declared (not defined) as pointer objects in the C code using the `extern` qualifier. Remember that the object `lookuptable` will not be initialised until this C function has been called and executed. Reading from the array before it is initialized will return incorrect values.

If you wish to have initialised objects copied to RAM before `main()` is executed, you can write assembler code, or copy and modify the appropriate routine in the run-time code that is supplied with the compiler. You can create your own run-time object file by pre-compiling the modified run-time file and using this object file instead of the standard file that is automatically linked with user's programs. From assembler, both the underscore characters are required when accessing the psect address symbols.

If you define your own psect based on a `bss` psect, then, in the same way, you will have to supply code to clear this area of memory if you are to assume that the objects defined within the psect will be cleared when they are first used.

2.3.4 Issues when linking

The linker uses a relatively complicated algorithm to relocate the psects contained in the object and library files passed to it, but the linker needs more information than that discussed above to know exactly how to relocate each psect? This information is contained in other the linker options passed to the linker by the driver and in the psect flags which are used with each psect directive. The following explain some of the issues the linker must take into account.

2.3.4.1 Paged memory

Let's assume that a processor has two ROM areas in which to place code and constant data. The linker will never split a psect over any memory boundary. A memory boundary is assumed to exist wherever there is a discontinuity in the address passed to the linker in the linker options. For example, if a class is specified using the addresses as follows:

```
-ADATA=0h-FFh,100h-1FFh
```

It is assumed that some boundary exists between address FFh and 100h, even though these addresses are contiguous. This is why you will see contiguous address ranges specified like this, instead of having one range covering the entire memory space. To make it easy to specify similar contiguous address ranges, a repeat count can be used, like:

```
-ADATA=0h-FFhx2
```

can be used; in this example, two ranges are specified: 0 to FFh and then 100h to 1FFh. Some processors have memory pages or banks. Again, a psect will not straddle a bank or page boundary.

Given that psects cannot be split over boundaries, having large psects can be a problem to relocate. If there are two, 1 kB areas of memory and the linker has to position a single 1.8 kB psect in this space, it will not be able to perform this relocation, even though the size of the psect is smaller than the total amount of memory available. The larger the psects, the more difficult it is for the linker to position them. If the above psect was split into three 0.6 kB psects, the linker could position two of them - one in each memory area - but the third would still not fit in the remaining space in either area. When writing code for processors like the PIC, which place the code generated from each C function into a separate, local psect, functions should not become too long.

If the linker cannot position a psect, it generates a "Can't find space for psect xxxx" error, where xxxx is the name of the psect. Remember that the linker relocates psects so it will not report memory errors with specific C functions or data objects. Search the assembler listing file to identify which C function is associated with the psect that is reported in the error message if local psects are generated by the code generator.

Global psects that are not overlaid are concatenated to form a single psect by the linker before relocation takes place. There are instances where this grouped psect appears to be split again to place it in memory. Such instances occur when the psect class within which it is a member covers several address ranges and the grouped psect is too large to fit any of the ranges. The linker may use intermediate groupings of the psects, called *clutches* to facilitate relocation within class address ranges. Clutches are in no way controllable by the programmer and a complete understanding of their nature is not required to be able to understand or use the linker options. It is suffice to say that global psects can still use the address ranges within a class. Note that although a grouped psect can be comprised of several clutches, an individual psect defined in a module can never be split under any circumstances.

2.3.4.2 Separate memory areas

Another issue faced by the linker is this: On some processors, there are distinct memory areas for program and data, i.e. Harvard architecture chips like the 8051XA. For example, ROM may extend from 0h - FFFFh and separate RAM may extend from 0h - 7FFh. If the linker is asked to position a psect at address 100h via a -p option, how does the linker know whether this is an address in program memory or in the data space? The linker makes use of the SPACE psect flag to determine this. Different areas are

assigned a different space value. For example ROM may be assigned a SPACE value of 0 and RAM a SPACE flag of 1. The space flags for each psect are shown in the map file.

The space flag is not used when the linker can distinguish the destination area of an object from its address. Some processors use memory banks which, from the processors's point of view, cover the same range of addresses, but which are within the same distinct memory area. In these cases, the compiler will assign unique addresses to objects in banked areas. For example, some PIC processors can access four banks of RAM, each bank covering addresses 0 to 7Fh. The compiler will assign objects in the first bank (bank 0) addresses 0 to 7Fh; objects in the second bank: 80h to FFh; objects in the third bank: 100h to 17Fh etc. This extra bank information is removed from the address before it is used in an assembler instruction. All PIC RAM banks use a SPACE flag of 1, but the ROM area on the PIC is entirely separate and uses a different SPACE flag (0). The space flag is not relevant to psects which reside in both memory areas, such as the data psects which are copied from ROM to RAM.

After relocation is complete, the linker will group psects together to form a *segment*. Segments, along with clutches, are rarely mentioned with the HI-TECH compiler simply because they are an abstract object used only by the linker during its operation. Segment details will appear in the map file. A segment is a collection of psects that are contiguous and which are destined for a specific area in memory. The name of a segment is derived from the name of the first psect that appears in the segment and should not be confused with the psect which has that name.

2.3.4.3 Objects at absolute addresses

After the psects have been relocated, the addresses of data objects can be resolved and inserted into the assembler instructions which make reference to an object's address. There is one situation where the linker does not determine and resolve the address of a C object. This is when the object has been defined as absolute in the C code. The following example shows the object DDRA being positioned at address 200h.

```
unsigned char DDRA @ 0x200;
```

When the code generator makes reference to the object DDRA, instead of using a symbol in the generated assembler code which will later be replaced with the object's address after psect relocation, it will immediately use the value 200h. The important thing to realise is that the instructions in the assembler that access this object will not have any symbols that need to be resolved, and so the linker will simply skip over them as they are already complete. If the linker has also been told, via its linker options, that there is memory available at address 200h for RAM objects, it may very well position a psect such that an object that resides in this psect also uses address 200h. As there is no symbol associated with the absolute object, the linker will not see that two objects are sharing the same memory. If objects are overlapping, the program will most likely fail unpredictably.

When positioning objects at absolute address, it vital to ensure that the linker will not position objects over those defined as absolute. Absolute objects are intended for C objects that are mapped over the top

of hardware registers to allow the registers to be easily access from the C source. The programmer must ensure that the linker options do not specify that there is any general-purpose RAM in the memory space taken up by the hardware. Ordinary variables to be positioned at absolute addresses should be done so using a separate psect (by simply defining your own using a psect directive in assembler code, or by using the `#pragma psect` directive in C code) and linker option to position the objects. If you must use an absolute address for an object in general-purpose RAM, make sure that the linker options are modified so that the linker will not position other psects in this area.

2.3.5 Modifying the linker options

In most applications, the default linker options do not need to be modified. It is recommended that if you think the options should be modified, but you do not understand how the linker options work, that you seek technical assistance in regard to the problem at hand.

If you do need to modify the linker options, there are several ways to do this. If you are simply adding another option to those present by default, the option can be specified to the CLD using a `-L` option. To position the `lut` psect that was used in the earlier example, the following option could be used.

```
-L-plut=500/const
```

The `-L` simply passes whatever follows to the linker. If you want to add another option to the default linker options and you are using HPD and a project, then it is a simple case of opening the **linker options...** dialogue box and adding the option to the end of those already there. The options should be entered exactly as they should be presented to the linker, i.e. you do not need the `-L` at the front.

If you want to modify existing linker options other than simply changing the memory address that are specified with the `-A` CLD option, then you cannot use the CLD to do this directly. What you will need to do is to perform the compilation and link separately. Let's say that the file `main.c` and `extra.c` are to be compiled for the 8051 with modified linker options. First we can compile up to, but not include, the link stage by using a command line something like this.

```
c51 -o -Zg -asmlist -C main.c extra.c
```

The `-C` options stops the compilation before the link stage. Include any other options which are normally required. This will create two object files: `main.obj` and `extra.obj`, which then have to be linked together.

Run the CLD again in verbose mode by giving a `-v` option on the command line, passing it the names of the object files created above, and redirect the output to a file. For example:

```
c51 -v -A8000,0,100,0,0 main.obj extra.obj > main.lnk
```

Note that if you do not give the `-A` CLD option, the compiler will prompt you for the memory addresses, but with the output redirected, you will not see the prompts.

The file generated (`main.lnk`) will contain the command line that CLD generated to run the linker with the memory values specified using the `-A` option. Edit this file and remove any messages printed by the compiler. Remove the command line for any applications run after the link stage, for example `objtohex` or `cromwell`, although you should take note of what these command lines are as you will need to run these applications manually after the link stage. The linker command line is typically very long and if a DOS batch file is used to perform the link stage, it is limited to lines 128 characters long. Instead the linker can be passed a command file which contains the linker options only. Break up the linker command line in the file you have created by inserting backslash characters `"\"` followed by a return. Also remove the name and path of the linker executable from the beginning of the command line so that only the options remain. The above command line generated a `main.lnk` file that was then edited as suggested above to give the following.

```
-z -pvector=08000h,text,code,data,const,strings \
-prbit=0/20h,rbss,rdata/strings,irdata,idata/rbss \
-pbss=0100h/idata -pnvram=bss,heap -ol.obj \
-m/tmp/06206eaa /usr/hitech/lib/rt51--ns.obj main.obj \
extra.obj /usr/hitech/lib/51--nsc.lib
```

Now, with care, modify the linker options in this file as required by your application.

Now perform the link stage by running the linker directly and redirecting its arguments from the command file you have created.

```
hlink < main.lnk
```

This will create an output file called `l.obj`. If other applications were run after the link stage, you will need to run them to generate the correct output file format, for example a HEX file.

Modifying the options to HPD is much simpler. Again, simply open the **linker options...** dialogue box and make the required changes, using the buttons at the bottom of the box to help with the editing. Save and re-make your project.

The map file will contain the command line actually passed to the linker and this can be checked to confirm that the linker ran with the new options.

2.4 Addresses used with the PIC

The PIC processor has a complicated memory map with *banked RAM* and *paged ROM*, and each memory having different word widths. One of the biggest sources of confusion regarding addresses used by the compiler stems from the fact that internal PIC RAM is one byte wide, but the ROM is either 12, 14 or 16 bits wide on baseline, midrange or highend processors, respectively. This tutorial explains the different addresses that are used by the HI-TECH compiler.

2.4.1 Code addresses

Labels used in PIC code (stored in ROM), such as C function names, are assigned *word addresses*. If you check the map file after a compilation, the addresses of C functions, assembler routines and labels shown in the symbol table are word addresses. The `text n` psects (where n is a number) are the psects which will contain the assembler generated from a C function. These psects use a DELTA psect flag of 2. This indicates that one addressable unit - in this case an instruction - will take up 2 bytes of memory. Even if the PIC has 12-bit wide ROM, the delta value is still 2. The delta value must be an integral number.

2.4.2 Data addresses

The addresses of data objects are a little more complicated. Consider an array of characters (bytes) that is placed in RAM. A label - the name of the array - will be used to reference the first element of the array. The array will be placed in one of the `rdata` or `rbss` psects depending on whether the array was initialised. Both these psects have a delta value of 1 which indicates that the addressable unit - in this case a character - is 1 byte long. The address of the array in RAM will be a *byte address*.

If the array was declared as `const`, then it will be stored in ROM. On baseline and midrange PICs, ROM cannot be read like RAM, so the values are stored as `retlw` (return with a literal (constant) in the W register) assembler instructions, each taking up one word of ROM. The instructions' data values represent each element of the array. Since the data is actually stored as code, the delta value of the psect containing the data will be 2 even though the object that these instructions represent is only a byte. The delta value must be 2 since it refers to the contents of the psect which is code, not byte-wide data values.

The highend PIC devices can read ROM memory directly and store the array elements as their byte values in the ROM. Highend PICs also have 16-bit wide ROM so the compiler can, and does, store 2 bytes for every word location in the ROM. Although these values are stored in a 16-bit wide area of memory, the delta value for the psect that contains the data is 1. This implies that the addressable unit in this area is one byte. This is necessary since you must be able to address each half of the word to access each of the array elements stored there.

The above has important implications for the addresses that result during linking of programs with constant data. With baseline and mid-range PICs, the address of a constant array label will be a word address; with highend PICs, it will be a byte address. If you want to search for the array in memory when using an emulator, for example, you must remember to divide the address by 2 to convert it to a word address if you are using a highend device. The map file for highend devices will also show byte addresses for the constant data, but the code labels will be word addresses. Sometimes it appears as if constant data has overwritten code since their addresses overlap, but again, you cannot compare byte and word addresses. Always check the delta value of a psect if you are unsure whether you are working with a byte or word address. The delta value is displayed with the psect's definition which can usually be seen in the assembler listing file for that module.

The delta value does not indicate the size of objects stored in the psect. For example, if the array had been a `const` array of `ints`, the delta value will still be 1, even though the size of an `int` is 2 bytes. The addressable unit of an `int` is 1, since it is possible (via pointers for example) to only access one half of the `int` value. On the PIC there are only two delta values: 2 for code, 1 for data.

As an example, consider the following do-nothing program.

```
const char array[] = {0x30, 0x31, 0x32};

void
main(void)
{
}
```

This is compiled for the 17C756 highend PIC processor. The assembler listing file includes the following information.

```

4          psect      cstrings,global,class=CODE,delta=1
5          psect      text0,local,class=CODE,delta=2
6          psect      text1,local,class=CODE,delta=2
...
26          psect cstrings
27 3FFC          _array
28 3FFC 0030          db 30
29 3FFD 0031          db 31
30 3FFE 0032          db 32
31
32          psect text0
33 1FFA          _main
34          ;const.c: 6: }
35 1FFA B020 0103 B000  ljmp start
+          0102
Symbol Table                                     Thu Jun 17 12:10:06 1999

_array 3FFC      _main 1FFA      start 2000
```

The array was placed in the `cstrings` psect which has a delta value of 1, but which was positioned in ROM since the object is `const`. The address of the array is listed as 3FFC. Since the delta value is 1, this must be a byte address. The routine `main` (label `_main`) is listed as starting at address 1FFA. Since

this routine is in the `text0` psect which has a delta value of 2, this is a word address. To determine where each psect is positioned relative to the other, convert the byte address to a word address (3FFC becomes 1FFE) and you will notice that the string is positioned immediately after the four-word main routine in ROM, which begins at 1FFA.

The HEX file generated was loaded into a simulator (MPLAB in this case) and the program memory examined. A section of this memory is shown below.

1FF9	FFFF		call	0x1FFF
<u>1FFA</u>	B020	main	movlw	0x20
1FFB	0103		movwf	0x3
1FFC	B000		movlw	0x0
1FFD	0102		movwf	0x2
<u>1FFE</u>	3130		cpfseq	0x30
1FFF	0032		ret	
2000	2B04	exit	setf	0x4

The three bytes representing the array's contents have been set in bold type. The address of the function `main` and the array are underlined in the address column. Notice that 2 characters have been stored at word address 1FFE, and the unused byte at the following address has been filled with a 0 byte. The mnemonic interpretation shown (the `cpfseq` and `ret` instructions) for the array is meaningless.

Note that the array label is not displayed in the code. Some simulators/emulators may indicate the label at the byte address, i.e. at an incorrect position. Unless the simulator/emulator knows about delta values and can adjust the addresses, some labels may not shown at the correct position. Note also, that often there can be several labels associated with an address. In the above example, the `__Lcstrings` label generated by the linker will also coincide with the name of the array since there is only one array in the `cstrings` psect. It can be difficult for the simulator/emulator to know which of these labels it should display.

If you check for objects in the HEX file remember that the addresses used in the records will be byte addresses. These will need to be divided by 2 if you are trying to find an object using a known word address.

Another thing that users must be aware of is that addresses used in linker options use the same units as the psects which they position. For example if you wished to position the `cstrings` psect at word address 1000h, you will need to use a linker option like the following.

```
-pcstrings=2000h
```

since the address is specified as a byte address (delta is 1 for the `cstrings` psect). But to position a psect which has a delta value of 2, you need to supply a word address.

The `cstrings` psect may be allocated an address using a class, for example the `ROMDATA` class. Each class is specified as using either byte or word addresses. `ROMDATA` uses word address. Thus, if `ROMDATA` is defined to cover the address range 8000h-BFFFh, and `cstrings` was positioned:

```
-pcstrings=ROMDATA
```

then the `cstrings` psect will be placed somewhere in the word address range 8000h to BFFFh and the map file will show that the `cstrings` psect was placed at a byte address somewhere between 10000h and 17FFEh. That is, the linker knows that the class uses word addresses and that the psect needs a byte address (since it has a delta value of 1) and has automatically scaled the addresses used.

The `CODE` class uses word addresses. To specify addresses for the `textn` psects which are in the `CODE` class (and have a delta value of 2), the linker does not need to scale any of the addresses. For example, to position the `textn` psects in the word address range 0 to 4FFh, the linker option is straightforward:

```
-ACODE=0-4FFh
```

and the map file will show the `textn` psect as residing somewhere in the word address range 0 to 4FFh.

The difference between the `cstrings` and `textn` psects is that the `textn` psects are in the `CODE` class, but that the `cstrings` psect is not in the `ROMDATA` class, even though it can use the `ROMDATA` class to specify an relocation address. To determine whether a class uses word or byte addresses, look at either the assembler listing for your program or the run-time assembler file to see what the delta value is for psects that are in the class in question.

2.4.3 Bit addresses

The other type of address that is sometimes seen are *bit addresses*. The compiler will produce a range of addresses that bit objects take up after compilation if there are any objects defined as type `bit` in the source. The addresses used in this case are bit addresses. Bit objects can only reside in RAM and are accessed via a special bit assembler instruction. The address used with this instruction is a byte address and a bit offset (the bit number within that byte). To convert a bit address to a byte address you simply divide it by 8 (the number of bits in a byte); the quotient is the byte address and the remainder is the bit offset. Bit psects are not usually explicitly position via a `-p` linker option since they are part of a class, but if they were positioned in this way, a bit address would be used.

Using HTLPIC

3.1 Introduction

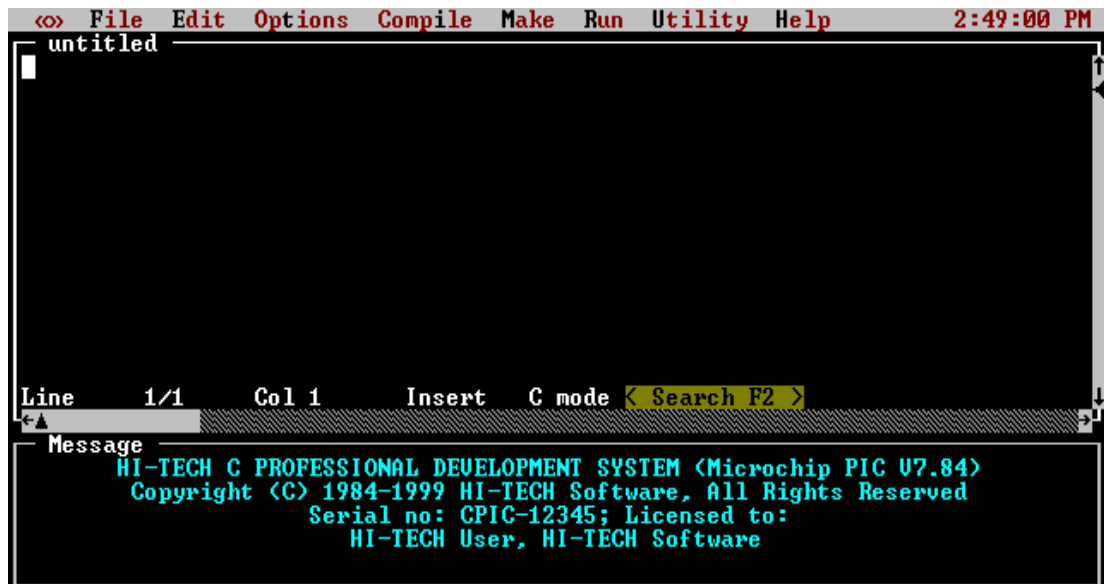
This chapter covers HTL, the **HI-TECH C Programmer's development Lite** environment. It assumes that you have already installed PIC LITE C.

HTLPIC is the version of HPL applicable to the PIC C compiler.

3.1.1 Starting HTLPIC

To start HTLPIC, simply type *htlpic* at the MS-DOS prompt. After a brief period of disk activity you will be presented with a screen similar to the one shown in Figure 3 - 1.

Figure 3 - 1 HTLPIC Startup Screen



The initial HTLPIC screen is broken into three windows. The top window contains the menu bar, the middle window the HTLPIC text editor, and the bottom window is the message window. Other windows may appear when certain menu items are selected. The editor window is what you will use most of the time.

HTLPIC uses the HI-TECH Windows user interface to provide a text screen-based user interface. This has multiple overlapping windows and pull-down menus. The user interface features which are common to all HI-TECH Windows applications are described later in this chapter.

Alternatively, HTLPIC can use a single command line argument. This is either the name of a text file, or the name of a *project file*. (Project files are discussed in a later section of this chapter.) If the argument has an extension *.prj*, HTLPIC will attempt to load a project file of that name. File names with any other extension will be treated as text files and loaded by the editor.

If an argument without an extension is given, HTLPIC will first attempt to load a *.prj* file, then a *.c* file. For example, if the current directory contains a file called *x.c* and HTLPIC is invoked with the command:

```
htlpic x
```

it will first attempt to load *x.prj* and when that fails, will load *x.c* into the editor. If no source file is loaded into the editor, an empty file with name *untitled* will be started.

3.2 The HI-TECH Windows User Interface

The HI-TECH Windows user interface used by HTLPIC provides a powerful text screen based user interface. This can be used through the keyboard alone, or with a combination of keyboard and mouse operations. For new users most operations will be simpler using the mouse, however, as experience with the package is gained, you will learn *hot-key* sequences for the most commonly used functions.

3.2.1 Environment variables

To use the HI-TECH C compiler, only one DOS environment variable need be present. This is a path to a temporary location where intermediate files may be stored. The variable is called TEMP and it should be automatically placed into you *autoexec.bat* file when the compiler is installed.

As this path is used to specify the location of temporary files, it should not be very long or the command lines that are generated to drive the compiler may exceed the DOS command line size limit. Typically C:\TEMP is chosen as the temporary file path.

3.2.2 Hardware Requirements

HI-TECH Windows based applications will run on any MS-DOS based machine with a standard display capable of supporting text screens of 80 columns by 25 rows or more. Higher resolution text modes like the EGA 80 x 43 mode will be recognised and used if the mode has already been selected before HTLPIC is executed. Higher modes can also be used with a */screen:xx* option as described below. Problems may be experienced with some poorly written VGA utilities. These may initialize the hardware to a higher resolution mode but leave the BIOS data area in low memory set to the values for an 80 x 25 display.

It is also possible to have HTLPIC set the screen display mode on EGA or VGA displays to show more than 25 lines. The option `/screen:xx` where `xx` is one of 25, 28, 43 or 50 will cause HTLPIC to set the display to that number of lines, or as close as possible. EGA display supports only 25 and 43 line text screens, while VGA supports 28 and 50 lines as well.

The display will be restored to the previous mode after HTLPIC exits. The selected number of lines will be saved in the `htlpic.ini` file and used for subsequent invocations of HTLPIC unless overridden by another `/screen` option.

HTLPIC will recognize and use any mouse driver which supports the standard INT 33H interface. Almost all modern mouse drivers support this standard device driver interface. Some older mouse drivers are missing a number of the driver status calls. If you are using such a mouse driver, HTLPIC will still work with the mouse, but the **Setup...** dialog in the `<>` menu will not work.

3.2.3 Colours

Colours are used in two ways in HTLPIC. First, there are colours associated with the screen display. These can be changed to suit your own preference. The second use of colour is to optionally code text entered into the text window. This assists you to see the different elements of a program as it is entered and compiled. These colours can also be changed to suit your requirements. Colours comprise two elements: the actual colour; and its attributes (such as bright or inverse). Table 3 - 1 on page 54 shows the colours and their values, whilst Table 3 - 2 on page 54 shows the attributes and their meaning.

Any colours are valid for the foreground but only colours 0 to 7 are valid for the background. Table 3 - 3 on page 55 shows the definition settings for the colours used by the editor when colour coding is selected.

The standard colour schemes for both the display colours and the text editor colour coding can be seen in the colour settings section of the `htlpic.ini` file. The first value in a colour definition is the foreground colour and the second is the background colour. To set the colours to other than the default sets you should remove the `#` before each line, then select the new colour value.

The `htlpic.ini` file also contains an example of an alternative standard colour scheme. The same process can be used to set the colour scheme for the menu bars and menus.

3.2.4 Pull-Down Menus

HI-TECH Windows includes a system of *pull-down menus* which operate from a *menu bar* across the top of the screen. The menu bar is broken into a series of words or symbols, each of which is the title of a single pull-down menu.

The menu system can be used with the keyboard, mouse, or a combination of mouse and keyboard actions. The keyboard and mouse actions that are supported are listed in Table 3 - 4 on page 55

Table 3 - 1 Colour values

Value	Colour
0	black
1	blue
2	green
3	cyan
4	red
5	magenta
6	brown
7	white
8	grey
9	bright blue
10	bright green
11	bright cyan
12	bright red
13	bright magenta
14	yellow
15	bright white

Table 3 - 2 Colour attributes

Attribute	description
normal:	normal text colour
bright:	bright/highlighted text colour
inverse:	inverse text colour
frame:	window frame colour
title:	window title colour
button:	colour for any buttons in a window

3.2.4.1 Keyboard Menu Selection

To select a menu item by keyboard press *alt-space* to open the menu system. Then use the arrow keys to move to the desired menu and highlight the item required. When the item required is highlighted select it by pressing *enter*. Some menu items will be displayed with lower intensity or a different colour and are not selectable. These items are disabled because their selection is not appropriate within the current context of the application. For example, the **Save project** item will not be selectable if no project has been loaded or defined.

Table 3 - 3 Colour coding settings

Setting	Description
C_wspace:	White space - foreground colour affects cursor
C_number:	Octal, decimal and hexadecimal numbers
C_alpha:	Alphanumeric variable, macro and function names
C_punct:	Punctuation characters etc.
C_keyword:	C keywords and variable types: eg int, static, etc.
C_brace:	Open and close braces: { }
C_s_quote:	Text in single quotes
C_d_quote:	Text in double quotes
C_comment:	Traditional C style comments: /* ... */
Cpp_comment	C++ style comments: // ...
C_preprocessor:	C pre-processor directives: #blah
Include_file:	Include file names
Error:	Errors - anything incorrect detected by the editor
Asm_code:	Inline assembler code (#asm...#endasm)
Asm-comment:	Assembler comments: ; ...

Table 3 - 4 Menu system key and mouse actions

Action	Key	Mouse
Open menu	Alt-space	Press left button in menu bar or press middle button anywhere in screen
Escape from menu	Alt-space or Escape	Press left button outside menu system displays
Select item	Enter	Release left or centre button on highlighted item or click left or centre button on an item
Next menu	Right arrow	Drag to right
Previous menu	Left arrow	Drag to left
Next item	Down arrow	Drag downwards
Previous item	Up arrow	Drag upwards

3.2.4.2 Mouse Menu Selection

To open the menu system, move the pointer to the title of the menu which you require and press the left button. You can browse through the menu system by holding the left button down and dragging the mouse across the titles of several menus, opening each in turn. You may also operate the menu system

with the middle button on three button mice. Press the middle button to bring the menu bar to the front. This makes it selectable even if it is completely hidden by a zoomed window.

Once a menu has been opened, two styles of selection are possible. If the left or middle button is released while no menu item is highlighted, the menu will be left open. Then you can select using the keyboard or by moving the pointer to the desired menu item and clicking the left or middle mouse button. If the mouse button is left down after the menu is opened, you can select by dragging the mouse to the desired item and releasing the button.

3.2.4.3 Menu Hot Keys

3

When browsing through the menu system you will notice that some menu items have *hot key* sequences displayed. For example, the HTLPIC menu item **Save** has the key sequence *alt-s* shown as part of the display. When a menu item has a key equivalent, it can be selected directly by pressing that key without opening the menu system. Key equivalents will be either *alt-alphanumeric* keys or *function keys*. Where function keys are used, different but related menu items will commonly be grouped on the one key. For example, in HTLPIC *F3* is assigned to **Compile and Link**, *shift-F3* is assigned to **Compile to .OBJ** and *ctrl-F3* is assigned to **Compile to .AS**.

Key equivalents are also assigned to entire menus, providing a convenient method of going to a particular menu with a single keystroke. The key assigned will usually be *alt* and the first letter of the menu name, for example *alt-e* for the **Edit** menu. The menu key equivalents are distinguished by being highlighted in a different colour (except monochrome displays) and are highlighted with inverse video when the *alt* key is depressed. A full list of HTLPIC key equivalents is shown in Table 3 - 5 on page 57.

3.2.5 Selecting windows

HI-TECH Windows allows you to overlap or tile windows. Using the keyboard, you can bring a window to the front by pressing *ctrl-enter* one or more times. Each time *ctrl-enter* is pressed, the rear-most window is brought to the front and the other windows shuffle one level towards the back. A series of *ctrl-enter* presses will cycle endlessly through the window hierarchy.

Using the mouse, you can bring any visible window to the front by pressing the left button in its content region¹. A window can be made rearmost by holding the *alt* key down and pressing the left button in its content region. If a window is completely hidden by other windows, it can usually be located either by pressing *ctrl-enter* a few times or by moving other windows to the back with *alt-left-button*.

Some windows will not come to the front when the left button is pressed in them. These windows have a special attribute set by the application and are usually made that way for a good reason. To give an example, the HTLPIC compiler error window will not be made front-most if it is clicked. Instead it will

1. Pressing the left mouse button in a window frame has a completely different effect, as discussed later in this chapter.

Table 3 - 5 HTLPIC menu hot keys

Key	Meaning
Alt-O	Open editor file
Alt-N	Clear editor file
Alt-S	Save editor file
Alt-A	Save editor file with new name
Alt-Q	Quit to DOS
Alt-J	DOS Shell
Alt-F	Open File menu
Alt-E	Open Edit menu
Alt-I	Open Compile menu
Alt-M	Open Make menu
Alt-R	Open Run menu
Alt-T	Open Options menu
Alt-U	Open Utility menu
Alt-H	Open Help menu
Alt-P	Open Project file
Alt-W	Warning level dialog
Alt-Z	Optimisation menu
Alt-D	Run DOS command
F3	Compile and link single file
Shift-F3	Compile to object file
Ctrl-F3	Compile to assembler code
Ctrl-F4	Retrieve last file
F5	Make target program
Shift-F5	Re-link target program
Ctrl-F5	Re-make all objects and target program
Shift-F7	User defined command 1
Shift-F8	User defined command 2
Shift-F9	User defined command 3
Shift-F10	User defined command 4
F2	Search in edit window
Alt-X	Cut to clipboard
Alt-C	Copy to clipboard
Alt-V	Paste from clipboard

accept the click as if it were already the front window. This allows the mouse to be used to select the compiler errors listed, while leaving the editor window at the front, so the program text can be altered.

3.2.6 Moving and Resizing Windows

Most windows can be moved and resized by the user. There is nothing on screen to distinguish windows which cannot be moved or resized. If you attempt to move or resize a window and nothing happens, it indicates that the window cannot be resized. Some windows can be moved but not resized, usually because their contents are of a fixed size and resizing would not make sense. The HTLPIC calculator is an example of a window which can be moved but not resized.

Windows can be moved and resized using the keyboard or the mouse. Using the keyboard, move/resize mode can be entered by pressing *ctrl-alt-space*. The application will respond by replacing the menu bar with the move/resize menu strip. This allows the front most window to be moved and resized. When the resizing is complete, press *enter* to return to the operating function of the window. A full list of all the operating keys is shown in Table 3 - 6.

Table 3 - 6 Resize mode keys

Key	Action
Left arrow	Move window to right
Right arrow	Move window to left
Up arrow	Move window upwards
Down arrow	Move window downwards
Shift-left arrow	Shrink window horizontally
Shift-right arrow	Expand window horizontally
Shift-up arrow	Shrink window vertically
Shift-down arrow	Expand window vertically
Enter or Escape	Exit move/resize mode

Move/resize mode can be exited with any normal application action, like a mouse click, pressing a hot key or activating the menu system by pressing *alt-space*. There are other ways of moving and resizing windows:

- ☐ Windows can be moved and resized using the mouse. You can move any visible window by pressing the left mouse button on its frame, dragging it to a new position and releasing the button. If a window is “grabbed” near one of its corners the pointer will change to a diamond. Then you can move the window in any direction, including diagonally. If a window is grabbed near the middle of the top or bottom edge the pointer will change to a vertical arrow. Then you can move the window vertically. If a window is grabbed near the middle of the left or right edge the pointer will change to a horizontal arrow. Then it will only be possible to move the window

horizontally.

- ☐ If a window has a *scroll bar* in its frame, pressing the left mouse button in the scroll bar will not move the window. Instead it activates the scroll bar, sending scroll messages to the application. If you want to move a window which has a frame scroll bar, just select a different part of the frame.
- ☐ Windows can be resized using the right mouse button. You can resize any visible window by pressing the right mouse button on its bottom or left frame. Then drag the frame to a new boundary and release the button. If a window is grabbed near its lower right corner the pointer changes to a diamond and it is possible to resize the window in any direction. If the frame is grabbed anywhere else on the bottom edge, it is only possible to resize vertically. If the window is grabbed anywhere else on the right edge it is only possible to resize horizontally. If the right button is pressed anywhere in the top or left edges nothing will happen.
- ☐ You can also *zoom* a window to its maximum size. The front most window can be zoomed by pressing *shift-(keypad)+*, if it is zoomed again it reverts to its former size. In either the zoomed or unzoomed state the window can be moved and resized. Zoom effectively toggles between two user defined sizes. You can also zoom a window by clicking the right mouse button in its content region.

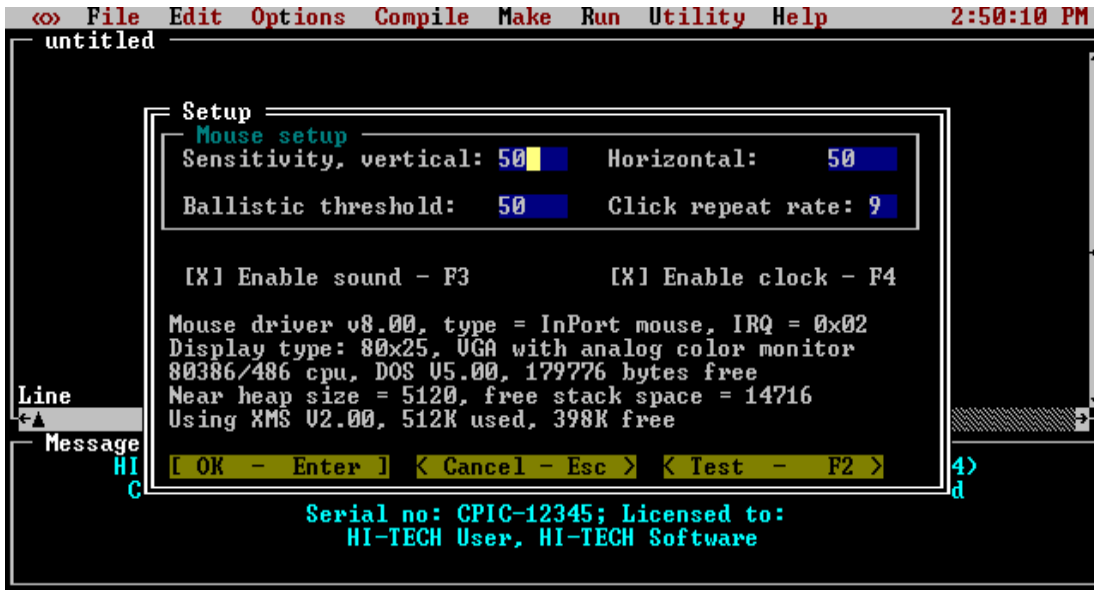
3.2.7 Buttons

Some windows contain *buttons* which can be used to select particular actions immediately. Buttons are like menu items which are always visible and selectable. A button can be selected either by clicking the left mouse button on it or by using its key equivalent. The key equivalent to a button will either be displayed as part of the button, or as part of a help message somewhere else in the window. For example, the HTLPIC error window (Figure 3 - 5 on page 64) contains a number of buttons, to select HELP you would either click the left mouse button on it or press *F1*.

3.2.8 The Setup menu

If you open the system menu, identified by the symbol <> on the menu bar, you will find two entries: the **About HTLPIC ...** entry, which displays information about the version number of HTLPIC; and the **Setup ...** entry. Selecting the Setup entry opens a dialog box as shown in Figure 3 - 2 on page 60. This box displays information about HTLPIC's memory usage, and allows you to set the mouse sensitivity, whether the time of day is displayed in the menu bar, and whether sound is used. After changing mouse sensitivity values, you can test them by clicking on the *Test* button. This will change the mouse values so you can test the altered sensitivity. If you subsequently click *Cancel*, they will be restored to the previous values. Selecting OK will confirm the altered values, and save them in HTLPIC's initialisation file, so they will be reloaded next time you run HTLPIC. The sound and clock settings will also be stored in the initialisation file if you select OK.

Figure 3 - 2 Setup Dialogue



3.3 Tutorial: Creating and Compiling a C Program

This tutorial should be sufficient to get you started using HTLPIC. It does not attempt to give you a comprehensive tour of HTLPIC's features - that is left to the reference section of this chapter. Even if you are an experienced C programmer but have not used a HI-TECH Windows-based application before, we strongly suggest that you complete this tutorial.

Before starting HTLPIC, you need to create a work directory. Make sure you are logged to the root directory on your hard disk and type the following commands:

```
C:\> md tutorial
C:\> cd tutorial
C:\> TUTORIAL> htlpic
```

You will be presented with the HTLPIC startup screen. At this stage, the editor is ready to accept whatever text you type. A flashing block cursor should be visible in the top left corner of the edit window. You are now ready to enter your first C program using HTLPIC.

```
#include<pic.h>
/*
* Demo program - flashes LEDs on
```



```

* Port B, responds to switch press
* on RA1. Usable on PICDEM board.
*/

#define PORTBIT(adr, bit) ((unsigned) (&adr)*8+(bit))
static bit button @ PORTBIT(PORTA, 1);

main(void)
{
    unsigned i;
    unsigned char j;
    TRISB = 0; /* all bits output */
    j = 0;

    for(;;) {
        PORTB = 0x00;      /* turn all on */
        for(i = 16000 ; --i ;)
            continue;
        PORTB = ~j;        /* output value of j */
        for(i = 16000 ; --i ;)
            continue;
        if(button == 0) /* if switch pressed, increment */
            j++;
    }
}

```

Type in the LED program, pressing *enter* once at the end of each line. You can enter blank lines by pressing *enter* without typing any text. Intentionally leave out the semi-colon at the end of the first line in main, as shown above:

Figure 3 - 3 shows the screen as it should appear after entry of the LED program.

You now have a C program (complete with one error!) entered and almost ready for compilation. All you need to do is save it to a disk file and then invoke the compiler. In order to save your source code to disk, you will need to select the **Save** item from the **File** menu (Figure 3 - 4 on page 63)

If you do not have a mouse, follow these steps:

- ☐ Open the menu system by pressing *alt-space*
- ☐ Move to the **Edit** menu using the *right arrow* key

Figure 3 - 3 LED Flashing program in HTLPIC

```

<> File Edit Options Compile Make Run Utility Help 2:54:01 PM
C:\TUTORIAL\LED.C
main(void)
{
    unsigned char    i
    unsigned char    j;

    TRISB = 0;                /* all bits output */
    j = 0;
    for(;;) {
        PORTB = 0x00;        /* turn all on */
        for(i = 100 ; --i ;>
            continue;

        PORTB = ~j;          /* output value of j */
        for(i = 100 ; --i ;>

```

Line 17/35 Col 1 Insert C mode < Search F2 >

Message

HI-TECH C PROFESSIONAL DEVELOPMENT SYSTEM (Microchip PIC U7.84)
 Copyright (C) 1984-1999 HI-TECH Software, All Rights Reserved
 Serial no: CPIC-12345; Licensed to:
 HI-TECH User, HI-TECH Software

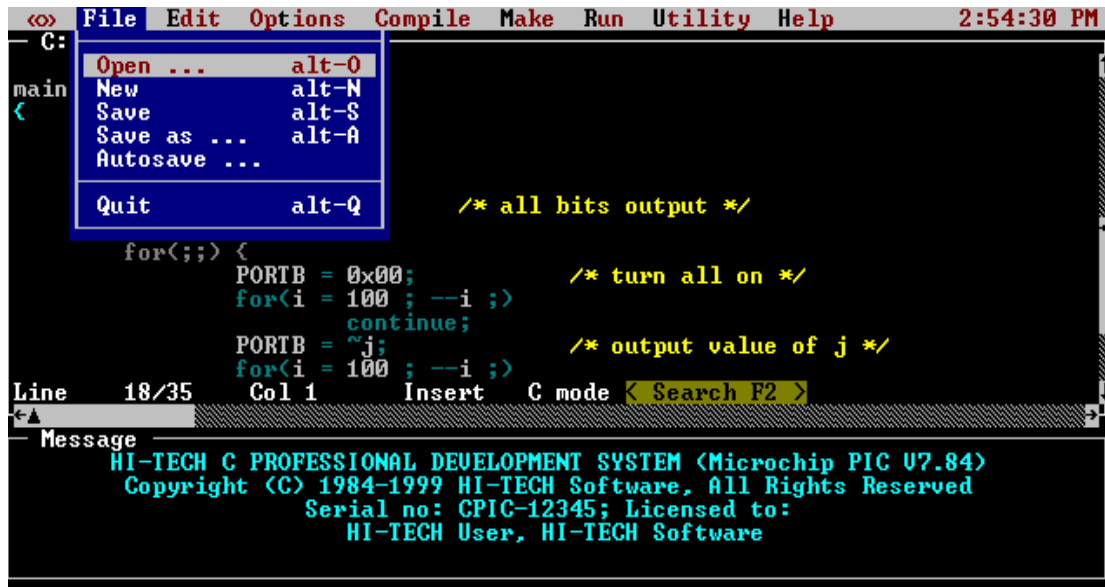
- ☐ Move down to the **Save** item using the *down arrow* key
- ☐ When the **Save** item is highlighted, select it by pressing the *enter* key.

If you are using the mouse, follow these steps:

- ☐ Open the **File** menu by moving the pointer to the word **File** in the menu bar and pressing the left button
- ☐ Highlight the **Save** item by dragging the mouse downwards with the left button held down, until the **Save** item is highlighted
- ☐ When the **Save** item is highlighted, select it by releasing the left button.

When the **File** menu (Figure 3 - 4) was open, you may have noticed that the **Save** item included the text *alt-s* at the right edge of the menu. This indicates that the save command can also be accessed directly using the *hot-key* command *alt-s*. A number of the most commonly used menu commands have hot-key equivalents which will either be alt-alphanumeric sequences or function keys.

Figure 3 - 4 HTLPIC File Menu



After **Save** has been selected, you should be presented with a dialog prompting you for the file name. If HTLPIC needs more information, such as a file name, before it is able to act on a command, it will always prompt you with a standard dialog.

The dialog contains an *edit line* where you can enter the file name to be used, and a number of *buttons*. These may be used to perform various actions within the dialog. A button may be selected by clicking the left mouse button with the pointer positioned on it, or by using its key equivalent. The text in the edit line may be edited using the standard editing keys: *left arrow*, *right arrow*, *backspace*, *del* and *ins*. *Ins* toggles the line editor between insert and overwrite mode.

In this case, save your C program to a file called *myled.c*. Type *myled.c* and then press *enter*. There should be a brief period of disk activity as HTLPIC saves the file.

You are now ready to actually compile the program. To compile and link in a single step, select the **Compile and link** item from the **Compile** menu, using the pull down menu system as before. Note that **Compile and link** has key *F3* assigned to it - in future you may wish to save time by using this key.

You will be asked to enter some information at this point:

Select Processor

Select the processor you are using.

Floating Point Type

Select 24-bit (default).

Optimisation

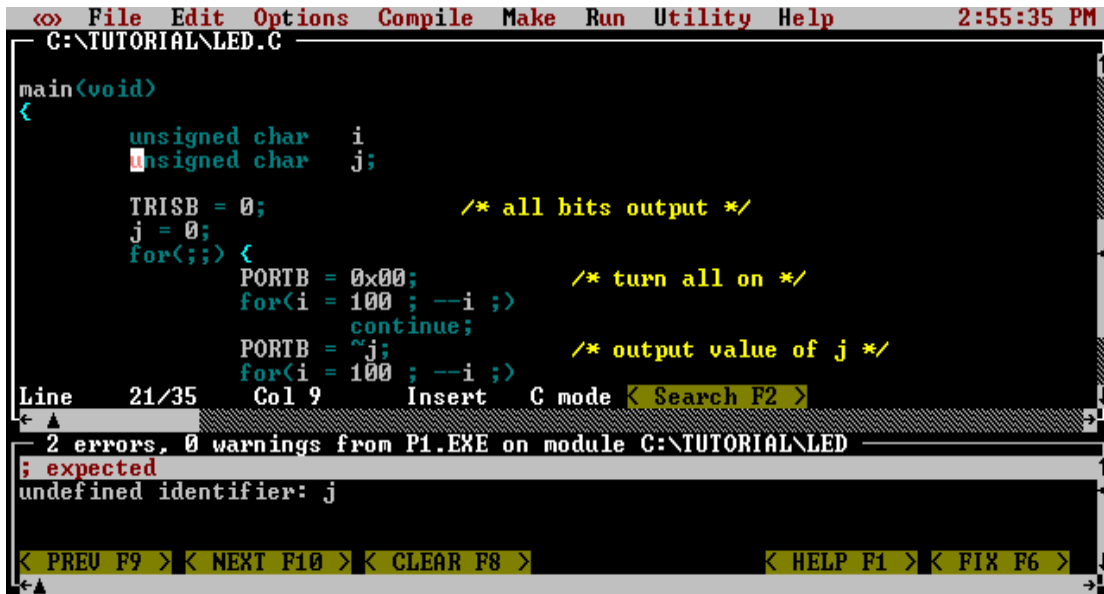
Select Full Optimisation, with the default Global Optimisation Level of 1

Output File Type

Select Bytecraft .COD or Intel .HEX.

This time, the compiler will not run to completion. This is because we deliberately omitted a semicolon on the end of a line, in order to see how HTLPIC handles compiler errors. After a couple of seconds of disk activity, you should hear a “splat” noise. The message window will be replaced by a window containing a number of buttons and describing two errors as shown in Figure 3 - 5 on page 64.

Figure 3 - 5 Error window



The screenshot shows a text editor window titled "C:\TUTORIAL\LED.C" with a menu bar (File, Edit, Options, Compile, Make, Run, Utility, Help) and a status bar (2:55:35 PM). The code in the editor is as follows:

```
main(void)
{
    unsigned char  i;
    unsigned char  j;

    TRISB = 0;                /* all bits output */
    j = 0;
    for(;;) {
        PORTB = 0x00;
        for(i = 100 ; --i ;> /* turn all on */
            continue;
        PORTB = ~j;          /* output value of j */
        for(i = 100 ; --i ;>
```

The status bar shows "Line 21/35 Col 9 Insert C mode < Search F2 >". Below the code, a message box displays "2 errors, 0 warnings from P1.EXE on module C:\TUTORIAL\LED". The error message is "; expected" and "undefined identifier: j". At the bottom, there are navigation buttons: "< PREV F9 >", "< NEXT F10 >", "< CLEAR F8 >", "< HELP F1 >", and "< FIX F6 >".

The text in the frame of the error window shows the number of compiler errors generated, and which phase of the compiler generated them. Most errors will come from *p1.exe* and *cglpic.exe*. *Cpp.exe* and *hlink.exe* can also return errors. In this case, the error window frame contains the message:

2 errors, 0 warnings from p1.exe

indicating that pass 1 of the compiler found 2 fatal errors, with the second error being caused by the first one. It is possible to configure HTLPIC so that non-fatal warnings will not stop compilation. If only warnings are returned, an additional button will appear, labelled CONTINUE. Selecting this button (or *F4*) will resume the compilation.

In this case, the error message *; expected* will be highlighted and the cursor will have been placed on the start of the line after the `unsigned i` declaration. This is where the error was first detected. The error window contains a number of buttons which allow you to select which error you wish to handle, clear the error status display, or obtain an explanation of the currently highlighted error. In order to obtain an explanation of the error message, either select the HELP button with a mouse click, or press *F1*.

The error explanation for the missing semi-colon does not give any more information than we already have. However, the explanations for some of the more unusual errors produced by the compiler can be very helpful. All errors produced by the pre-processor (*cpp*), pass 1 (*p1*), code generator (*cglpic*), assembler (*aspic*) and linker (*hlink*) are handled. You may dismiss the error explanations by selecting the HIDE button (press *escape* or use the mouse).

In this instance HTLPIC has analysed the error, and is prepared to fix the error itself. This is indicated by the presence of the *FIX* button in the bottom right-hand corner of the error window. If HTLPIC is unable to analyse the error, it will not show the *FIX* button. Clicking on the *FIX* button, or pressing *F6* will fix the error by adding a semicolon to the end of the previous line. A “bip-bip” sound will be generated, and if there was more than one error line in the error window, HTLPIC will move to the next error.

To manually correct the error, move the cursor to the end of the declaration and add the missing semi-colon. If you have a mouse, simply click the left button on the position to which you want to move the cursor. If you are using the keyboard, move the cursor with the arrow keys. Once the missing semi-colon has been added, you are ready to attempt another compilation.

This time, we will “short circuit” the edit-save-compile cycle by pressing *F3* to invoke the **Compile and link** menu item. HTLPIC will automatically save the modified file to a temporary file, then compile it. The message window will then display the commands issued to each compiler phase in turn. If all goes well, you will hear a tone and see the message *Compilation successful*.

This tutorial has presented a simple overview of single file edit/compile development. HTLPIC is also capable of supporting multi-file projects (including mixed C and assembly language sources) using the project facility. The remainder of this chapter presents a detailed reference for the HTLPIC menu system, editor and project facility.

3.4 The HTLPIC editor

HTLPIC has a built-in text editor designed for the creation and modification of program text. The editor is loosely based on *WordStar* with a few minor differences and some enhancements for mouse-based

operation. If you are familiar with WordStar or any similar editor you should be able to use the HTLPIC editor without further instruction. HTLPIC also supports the standard PC keys, and thus should be readily usable by anyone familiar with typical MS-DOS or Microsoft Windows editors.

The HTLPIC editor is based in its own window, known as the *edit window*. The edit window is broken up into three areas, the *frame*, the *content region* and the *status line*.

3.4.1 Frame

The *frame* indicates the boundary between the edit window and the other windows on the desktop. The name of the current edit file is displayed in the top left corner of the frame. If a newly created file is being edited, the file name will be set to “untitled”. The frame can be manipulated using the mouse, allowing the window to be moved around the desktop and re-sized.

3.4.2 Content Region

The *content region*, which forms the largest portion of the window, contains the text being edited. When the edit window is active, the content region will contain a cursor indicating the current insertion point. The text in the content region can be manipulated using keyboard commands alone, or a combination of keyboard commands and mouse actions. The mouse can be used to position the cursor, scroll the text and select blocks for clipboard operations.

3.4.3 Status Line

The bottom line of the edit window is the *status line*. It contains the following information about the file being edited:

- ☐ *Line* shows the current line number, counting from the start of the file, and the total number of lines in the file.
- ☐ *Col* shows the number of the column containing the cursor, counting from the left edge of the window.
- ☐ If the status line includes the text *^K* after the *Col* entry, it indicates that the editor is waiting for the second character of a
- ☐ WordStar *ctrl-k* command. See the section Keyboard Commands on page 67, for a list of the valid *ctrl-k* commands.
- ☐ If the status line includes the text *^Q* after the *Col* entry, the editor is waiting for the second character of a WordStar *ctrl-q* command. See the section Keyboard Commands on page 67, for a list of the valid *ctrl-q* commands.
- ☐ *Insert* indicates that text typed on the keyboard will be inserted at the cursor position. Using the *insert mode toggle* command (the *ins* key on the keypad, or *ctrl-v*), the mode can be toggled between *Insert* and *Overwrite*. In overwrite mode, text entered on the keyboard will overwrite

characters under the cursor, instead of inserting them before the cursor.

- ☐ *Indent* indicates that the editor is in auto-indent mode. Auto-indent mode is toggled using the *ctrl-q i* key sequence. By default, auto-indent mode is enabled. When auto-indent mode is enabled, every time you add a new line the cursor is aligned under the first non-space character in the preceding line. If the file being edited is a C file, the editor will default to *C mode*. In this mode, when an opening brace ("*{*") is typed, the next line will be indented one tab stop. In addition, it will automatically align a closing brace ("*}*") with the first non-blank character on the line containing the opening brace. This makes the auto-indent mode ideal for entering C code.
- ☐ The **SEARCH** button may be used to initiate a search operation in the editor. To select **SEARCH**, click the left mouse button anywhere on the text of the button. The search facility may also be activated using the *F2* key and the WordStar *ctrl-q f* sequence.
- ☐ The **NEXT** button is only present if there has already been a search operation. It searches forwards for the next occurrence of the search text. **NEXT** may also be selected using *shift-F2* or *ctrl-l*.
- ☐ The **PREVIOUS** button is used to search for the previous occurrence of the search text. This button is only present if there has already been a search operation. The key equivalents for **PREVIOUS** are *ctrl-F2* and *ctrl-p*.

3.4.4 Keyboard Commands

The editor accepts a number of keyboard commands, broken up into the following categories:

- ☐ *Cursor movement* commands
- ☐ *Insert/delete* commands
- ☐ *Search* commands
- ☐ *Block and Clipboard* operations and
- ☐ *File* commands.

Each of these categories contains a number of logically related commands. Some of the cursor movement commands and block selection operations can also be performed with the mouse.

Table 3 - 7, "Editor keys," on page 68 provides an overview of the available keyboard commands and their key mappings. A number of the commands have multiple key mappings, some also have an equivalent menu item.

The **Zoom** command, *ctrl-q z*, is used to toggle the editor between windowed and full-screen mode. In full screen mode, the HTLPIC menu bar may still be accessed either by pressing the *alt* key or by using the middle button on a three button mouse.

Table 3 - 7 Editor keys

Command	Key	WordStar key
Character left	left arrow	Ctrl-S
Character right	right arrow	Ctrl-D
Word left	Ctrl-left arrow	Ctrl-A
Word right	Ctrl-right arrow	Ctrl-F
Line up	up arrow	Ctrl-E
Line down	down arrow	Ctrl-X
Page up	PgUp	Ctrl-R
Page down	PgDn	Ctrl-C
Start of line	Home	Ctrl-Q S
End of line	End	Ctrl-Q D
Top of window		Ctrl-Q E
Bottom of window		Ctrl-Q X
Start of file	Ctrl-Home	Ctrl-Q R
End of file	Ctrl-End	Ctrl-Q C
Insert mode toggle	Ins	Ctrl-V
Insert CR at cursor		Ctrl-N
Open new line below cursor		Ctrl-O
Delete char under cursor	Del	Ctrl-G
Delete char to left of cursor	Backspace	Ctrl-H
Delete line		Ctrl-Y
Delete to end of line		Ctrl-Q Y
Search	F2	Ctrl-Q F
Search forward	Shift-F2	Ctrl-L
Search backward	Alt-F2	Ctrl-P
Toggle auto indent mode		Ctrl-Q I
Zoom or unzoom window		Ctrl-Q Z
Open file	Alt-O	
New file	Alt-N	
Save file	Alt-S	
Save file - New name	Alt-A	

3.4.5 Block Commands

In addition to the movement and editing command listed in the “Editor Keys” table, the HTLPIC editor also supports WordStar style block operations and mouse driven cut/copy/paste clipboard operations.

The clipboard is implemented as a secondary editor window, allowing text to be directly entered and edited in the clipboard. The WordStar style block operations may be freely mixed with mouse driven clipboard and cut/copy/paste operations.

The block operations are based on the *ctrl-k* and *ctrl-q* key sequences which are familiar to anyone who has used a WordStar compatible editor.

Table 3 - 8, “Block operation keys,” on page 69 lists the WordStar compatible block operations which are available.

Table 3 - 8 Block operation keys

Command	Key sequence
Begin block	Ctrl-K B
End block	Ctrl-K K
Hide or show block	Ctrl-K H
Go to block start	Ctrl-Q B
Go to block end	Ctrl-Q K
Copy block	Ctrl-K C
Move block	Ctrl-K V
Delete block	Ctrl-K Y
Read block from file	Ctrl-K R
Write block to file	Ctrl-K W

The block operations behave in the usual manner for WordStar type editors with a number of minor differences. “Backwards” blocks, with the block end before the block start, are supported and behave exactly like a normal block selection. If no block is selected, a single line block may be selected by keying block-start (*ctrl-k b*) or block-end (*ctrl-k k*). If a block is already present, any block start or end operation has the effect of changing the block bounds.

Begin Block

ctrl-k b

The key sequence *ctrl-k b* selects the current line as the start of a block. If a block is already present, the block start marker will be shifted to the current line. If no block is present, a single line block will be selected at the current line.

End Block

ctrl-k k

The key sequence *ctrl-k k* selects the current line as the end of a block. If a block is already present, the block end marker will be shifted to the current line. If no block is present, a single line block will be selected at the current line.

Go To Block Start

ctrl-q b

If a block is present, the key sequence *ctrl-q b* moves the cursor to the line containing the block start marker.

Go To Block End

ctrl-q k

If a block is present, the key sequence *ctrl-q k* moves the cursor to the line containing the block end marker.

Block Hide Toggle

ctrl-k h

The block hide/display toggle, *ctrl-k h* is used to hide or display the current block selection. Blocks may only be manipulated with cut, copy, move and delete operations when displayed. The bounds of hidden blocks are maintained through all editing operations so a block may be selected, hidden and re-displayed after other editing operations have been performed. Note that some block and clipboard operations change the block selection, making it impossible to re-display a previously hidden block.

Copy Block

ctrl-k c

The *ctrl-k c* command inserts a copy of the current block selection before the line which contains the cursor. A copy of the block will also be placed in the clipboard. This operation is equivalent to a clipboard *Copy* operation followed by a clipboard *Paste* operation.

Move Block

ctrl-k v

The *ctrl-k v* command inserts the current block before the line which contains the cursor, then deletes the original copy of the block. That is, the block is moved to a new position just before the current line. A copy of the block will also be placed in the clipboard. This operation is equivalent to a clipboard *Cut* operation followed by a clipboard *Paste* operation.

Delete Block

ctrl-k y

The *ctrl-k y* command deletes the current block. A copy of the block will also be placed in the clipboard. This operation may be undone using the clipboard *Paste* command. This operation is equivalent to the clipboard *Cut* command.

Read block from file

ctrl-k r

The *ctrl-k r* command prompts the user for the name of a text file which is to be read and inserted before the current line. The inserted text will be selected as the current block. This operation may be undone by deleting the current block.

Write block to file**ctrl-k w**

The *ctrl-k w* command prompts the user for the name of a text file to which the current block selection will be written. This command does not alter the block selection, editor text or clipboard in any way.

Indent

This operation is available via the *Edit* menu. It will indent by one tab stop, the current block or the current line if no block is selected.

Outdent

This is the opposite of the previous operation, i.e. it removes one tab from the beginning of each line in the selection, or the current line if there is no block selected. It is only accessible via the *Edit* menu.

Comment/Uncomment

Also available in the *Edit* menu, this operation will insert or remove a C++ style comment leader (//) at the beginning of each line in the current block, or the current line if there is no block selected. If a line is currently uncommented, it will be commented, and if it is already commented, it will be uncommented. This is repeated for each line in the selection. This allows a quick way of commenting out a portion of code during debugging or testing.

3.4.6 Clipboard Editing

The HTLPIC editor also supports mouse driven clipboard operations, similar to those supported by several well known graphical user interfaces.

Text may be selected using mouse click and drag operations, deleted, cut or copied to the clipboard, and pasted from the clipboard. The clipboard is based on a standard editor window and may be directly manipulated by the user. Clipboard operations may be freely mixed with WordStar style block operations.

3.4.6.1 Selecting Text

Blocks of text may be selected using left mouse button and click or drag operations. The following mouse operations may be used:

- ☐ A single click of the left mouse button will position the cursor and hide the current selection. The **Hide** menu item in the **Edit** menu, or the *ctrl-k h* command, may be used to re display a block selection which was cancelled by a mouse click.
- ☐ A double click of the left mouse button will position the cursor and select the line as a single line block. Any previous selection will be cancelled.
- ☐ If the left button is pressed and held, a multi line selection from the position of the mouse click may be made by dragging the mouse in the direction which you wish to select. If the mouse moves outside the top or bottom bounds of the editor window, the editor will scroll to allow a selection of more than one page to be made. The cursor will be moved to the position of the

mouse when the left button is released. Any previous selection will be cancelled.

3.4.6.2 Clipboard Commands

The HTLPIC editor supports a number of clipboard manipulation commands which may be used to cut text to the clipboard, copy text to the clipboard, paste text from the clipboard, delete the current selection and hide or display the current selection. The clipboard window may be displayed and used as a secondary editing window. A number of the clipboard operations have both menu items and *hot key* sequences. The following clipboard operations are available:

Cut alt-x

The Cut option copies the current selection to the clipboard and then deletes the selection. This operation may be undone using the *Paste* operation. The previous contents of the clipboard are lost.

Copy alt-c

The Copy option copies the current selection to the clipboard without altering or deleting the selection. The previous contents of the clipboard are lost.

Paste alt-v

The Paste option inserts the contents of the clipboard into the editor before the current line. The contents of the clipboard are not altered.

Hide ctrl-k h

The Hide option toggles the current selection between the hidden and displayed state. This option is equivalent to the WordStar *ctrl-k h* command.

Show clipboard

This menu options hides or displays the clipboard editor window. If the clipboard window is visible, it is hidden. If the clipboard window is hidden it will be displayed and selected as the current window. The clipboard window behaves like a normal editor window in most respects except that no block operations may be used. This option has no key equivalent.

Clear clipboard

This option clears the contents of the clipboard, and cannot be undone. If a large selection is placed in the clipboard, you should use this option to make extra memory available to the editor after you have completed your clipboard operations.

Delete selection

This menu option deletes the current selection without copying it to the clipboard. Delete selection should not be confused with *Cut* as it cannot be reversed and no copy of the deleted text is kept. Use this option if you wish to delete a block of text without altering the contents of the clipboard.

3.5 HTLPIC menus

This section presents a item-by-item description of each of the HTLPIC menus. The description of each menu includes a screen print showing the appearance of the menu within a typical HTLPIC screen.

3.5.1 <>> menu

The <>> (system) menu is present in all HI-TECH Windows based applications. It contains handy system configuration utilities and *desk accessories* which we consider worth making a standard part of the desktop.

About HTLPIC ...

The About HTLPIC dialog displays information on the version number of the compiler and the licence details.

Setup ...

This menu item selects the standard mouse firmware configuration menu, and is present in all HI-TECH Windows based applications. The “mouse setup” dialog allows you to adjust the horizontal and vertical sensitivity of the mouse, the *ballistic threshold*² of the mouse and the mouse button auto-repeat rate.

This menu item will not be selectable if there is no mouse driver installed. With some early mouse drivers, this dialog will not function correctly. Unfortunately there is no way to detect drivers which exhibit this behaviour, because even the “mouse driver version info” call is missing from some of the older drivers!

This dialog will also display information about what kind of video card and monitor you have, what DOS version is used and free DOS memory available. See Figure 3 - 2 on page 60

3.5.2 File menu

The **File** menu contains file handling commands, the HTLPIC **Quit** command and the pick list:

Open ...

alt-O

This command loads a file into the editor. You will be prompted for the file name and if a wildcard (e.g. “*.C”) is entered, you will be presented with a file selector dialog. If the previous edit file has been modified but not saved, you will be given an opportunity to save it or abort the Open command.

New

alt-N

The **New** command clears the editor and creates a new edit file with default name “untitled”. If the previous edit file has been modified but not saved, you will be given a chance to save it or abort the New command.+

2. The ballistic threshold of a mouse is the speed beyond which the response of the pointer to further movement becomes exponential. Some primitive mouse drivers do not support this feature.

Save

alt-S

This command saves the current edit file. If the file is “untitled”, you will be prompted for a new name, otherwise the current file name (displayed in the edit window's frame) will be used.

Save as ...

alt-A

This command is similar to **Save**, except that a new file name is always requested.

Autosave ...

This item will invoke a dialog box allowing you to enter a time interval in minutes for auto saving of the edit file. If the value is non-zero, then the current edit file will automatically be saved to a temporary file at intervals. Should HTLPIC not exit normally, e.g. if your computer suffers a power failure, the next time you run HTLPIC, it will automatically restore the saved version of the file.

Quit

alt-Q

The **Quit** command is used to exit from HTLPIC to the operating system. If the current edit file has been modified but not saved, you will be given an opportunity to save it or abort the Quit command.

Clear pick list

This clears the list of recently-opened files which appear below this option.

Pick list

ctrl-F4

The pick list contains a list of the most recently-opened files. A file may be loaded from the pick list by selecting that file. The last file that was open may be retrieved by using the short-cut *ctrl-F4*.

3.5.3 Edit menu

The Edit menu contains items relating to the text editor and clipboard. The edit menu is shown in Figure 3 - 6.

Cut

alt-X

The **Cut** option copies the current selection to the clipboard and then deletes the selection. This operation may be undone using the **Paste** operation. The previous contents of the clipboard are lost.

Copy

alt-C

The **Copy** option copies the current selection to the clipboard without altering or deleting the selection. The previous contents of the clipboard are lost.

Paste

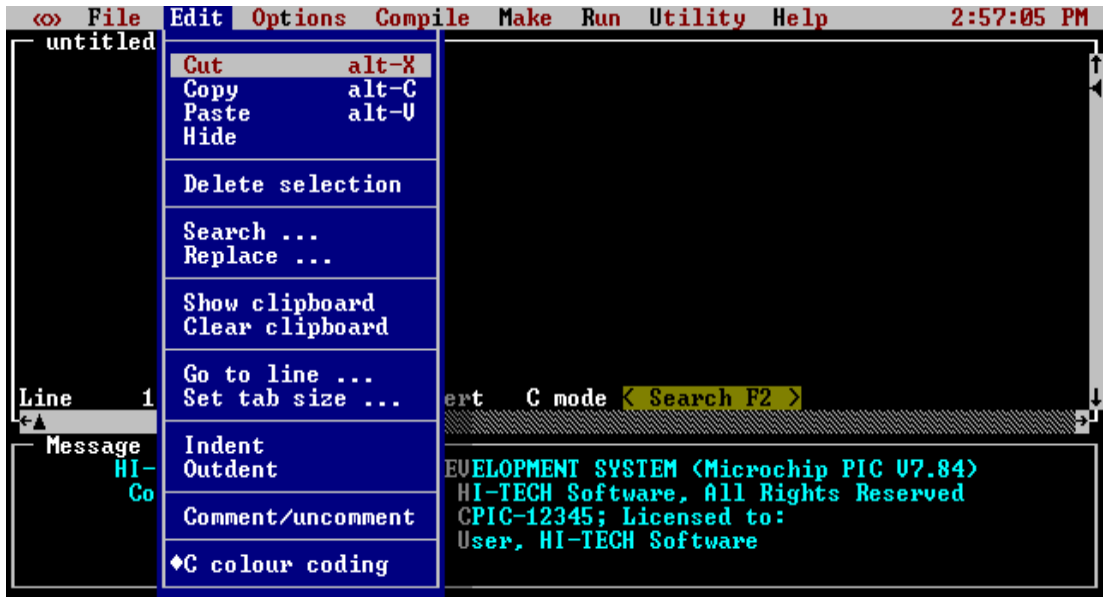
alt-V

The **Paste** option inserts the contents of the clipboard into the editor before the current line. The contents of the clipboard are not altered.

Hide

The **Hide** option toggles the current selection between the hidden and displayed state. This option is equivalent to the WordStar **ctrl-K H** command.

Figure 3 - 6 HTLPIC Edit Menu

**Delete selection**

This menu option deletes the current selection without copying it to the clipboard. **Delete selection** should not be confused with **Cut** as it cannot be reversed and no copy of the deleted text is kept. Use this option if you wish to delete a block of text without altering the contents of the clipboard.

Search ...

This option produces a dialog to allow you to enter a string for a search. You can select to search forwards or backwards by selecting the appropriate button. You can also decide if the search should be case sensitive and if a replacement string is to be substituted. You make these choices by clicking in the appropriate brackets.

Replace ...

This option is almost the same as the search option. It is used where you are sure you want to search and replace in the one operation. You can choose between two options. You can search and then decide whether to replace each time the search string is found. Alternatively, you can search and replace globally. If the global option is chosen, you should be careful in defining the search string as the replace can not be undone.

Show clipboard

This menu option hides or displays the clipboard editor window. If the clipboard window is already visible, it will be hidden. If the clipboard window is currently hidden it will be displayed and selected as the current window. The clipboard window behaves like a normal editor window in most respects except that no block operations may be used. This option has no key equivalent.

Clear clipboard

This option clears the contents of the clipboard, and cannot be undone. If a large selection is placed in the clipboard, you should use this option to make extra memory available to the editor after you have completed your clipboard operations.

Go to line ...

The **Go to line** command allows you to go directly to any line within the current edit file. You will be presented with a dialog prompting you for the line number. The title of the dialog will tell you the allowable range of line numbers in your source file.

Set tab size ...

This command is used to set the size of tab stops within the editor. The default tab size is 8, values from 1 to 16 may be used. For normal C source code 4 is also a good value. The tab size will be stored as part of your project if you are using the *Make* facility.

Indent

Selecting this item will indent by one tab stop the currently highlighted block, or the current line if there is no block selected.

Outdent

This is the reverse operation to Indent. It removes one tab from the beginning of each line in the currently selected block, or current line if there is no block.

Comment/Uncomment

This item will insert or remove C++ style comment leaders (*//*) from the beginning of each line in the current block, or the current line. This has the effect of commenting out those lines of code so that they will not be compiled. If a line is already commented in this manner, the comment leader will be removed.

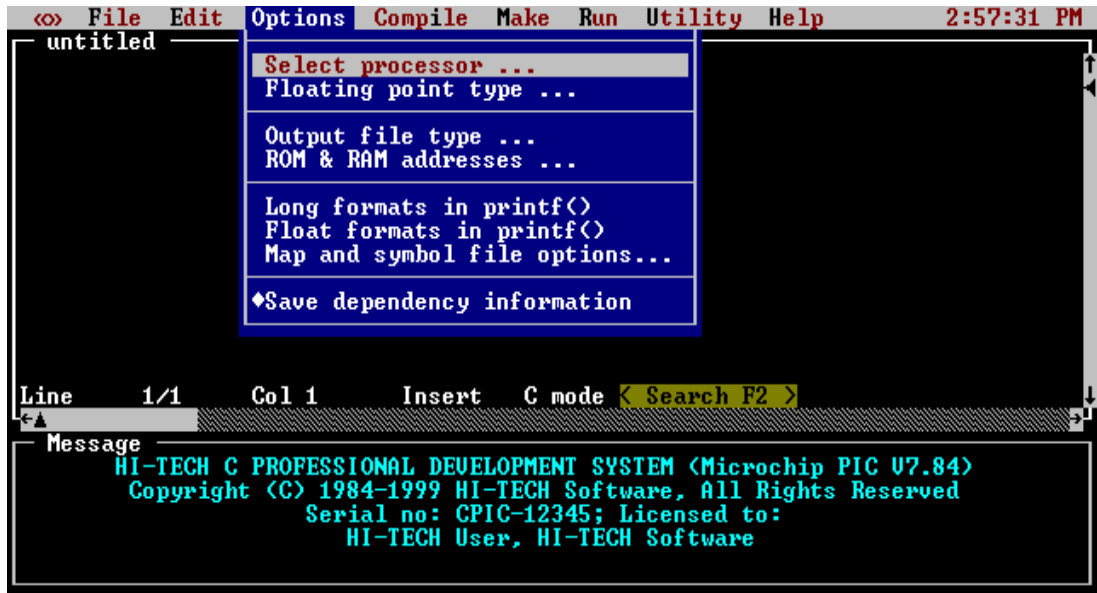
C colour coding

This option toggles the colour coding of text in the editor window. It turns on and off the colours for the various types of text. A mark appears before this item when it is active. For a full description of colours used in HTLPIC and how to select specific schemes, you should refer to the section, Colours on page 53.

3.5.4 Options menu

The **Options** menu contains commands which allow selection of compiler options, and target processor. Selections made in this menu will be stored in a project file, if one is being used. The Options menu is shown in Figure 3 - 7 on page 77..

Figure 3 - 7 Options Menu

**Select processor ...**

This option activates a dialog box which allows you to select the processor type you wish to use. The help string at the bottom of the dialogue indicates the amount of ROM space, in words, that each device contains. In addition the string indicates the number of RAM banks the device contains and the total amount of general-purpose RAM bytes available. This is the amount of RAM other than that used by the special function registers. If the help string displays "including common" then some of the RAM space is taken up by common memory which is used by the compiler for internal use.

Floating point type ...

This selects the format used for floating point doubles. The default format is the 24-bit truncated IEEE 754 format, but you may choose to use the 32-bit IEEE 754 format. For more information on these formats, see Floating Point on page 114.

Output file type ...

The default output file type is Bytecraft COD. The other choices are: Motorola S-Record HEX, Intel HEX, Binary Image, UBROF, Tektronix HEX, American Automation symbolic HEX and Intel OMF-51. This option will also allow you to specify that you want to create a library. A library can only be created from a project file.

ROM addresses ...

This menu is highlighted when a high-end PIC device is selected from the Select processor menu. This dialog allows you to specify the address ranges covered by external ROM devices. The addresses are used to store code and const data. (This option is disabled under PIC LITE)

Map and symbol file options ...

This dialog box allows you to set various options pertaining to debug information, the map file and the symbol file.

Source level debug info

This menu item is used to enable or disable source level debug information in the current symbol file. If you are using MPLAB, you should enable this option

Sort map by address

By default, the symbol table in the in the link map will be sorted by name. This option will cause it to be sorted numerically, based on the value of the symbol.

Suppress local symbols

Prevents the inclusion of all local symbols in the symbol file. Even if this option is not active, the linker will filter irrelevant compiler generated symbols from the symbol file.

Fake local symbols

This modifies the debug information produced to allow MPLAB to examine most local variables. It also adjusts source-level single stepping information to be that required by MPLAB. See also Section 5.36 on page 142.

MPLAB-ICD support

This button automatically adjusts the linker settings so that the output code is suitable for the MPLAB In-Circuit Debugger. This menu item is only highlighted if the selected processor has ICD capability.

Save dependency information

With this checked (which is the default), dependency information is saved in the project file. This means that restarting the HTL is much faster for a large project.

3.5.5 Compile menu

The **Compile** menu, shown in Figure 3 - 8 on page 79, contains the various forms of the compile command along with several machine independent compiler configuration options.

Compile and link

This command will compile a single source file and then invoke the linker and other utilities to produce an executable file. If the source file is an *.as* file, it will be passed directly to the assembler. The output file will have the same base name as the source file, but a different extension. For example *led.c* would be compiled to *led.cod*.

F3

Figure 3 - 8 HTLPIC Compile Menu

**Compile to .OBJ****shift-F3**

Compiles a single source file to a *.obj* file only. The linker and objtohex are not invoked. The *.as* files will be passed directly to the assembler. The object file produced will have the same base name as the source file and the extension *.obj*.

Compile to .AS**ctrl-F3**

This menu item compiles a single source file to assembly language, producing an assembler file with the same base name as the source file and the extension *.as*. This option is handy if you want to examine or modify the code generated by the compiler. If the current source file is an *.as* file, nothing will happen.

Preprocess only to .PRE

This runs the source file through the C preprocessor. The output of this action is a *.pre* file of the same name as the source file.

Stop on Warnings

This toggle determines whether compilation will be halted when non-fatal errors are detected. A mark appears against this item when it is active.

Warning level ...

alt-W

This command calls up a dialog which allows you set the compiler warning level, i.e. it determines how selective the compiler is about legal but dubious code. The range of currently implemented warning levels is -9 to 9, where lower warning levels are stricter. At level 9 all warnings (but not errors) are suppressed. Level 1 suppresses the *func() declared implicit int* message which is common when compiling UNIX-derived code. Level 3 is suggested for compiling code written with less strict (and K&R) compilers. Level 0 is the default. This command is equivalent to the -W option of the PICL command.

Optimisation ...

alt-Z

Selecting this item will open a dialog allowing you to select different kinds and levels of optimisation. The default is no optimization. Selections made in this dialog will be saved in the project file if one is being used.

Identifier length...

By default C identifiers are considered significant only to 31 characters. This command will allow setting the number of significant characters to be used, between 31 and 255.

Disable non-ANSI features

This option is used to enable strict ANSI conformance of all special keywords. HI-TECH C supports the special keywords such as *persistent* and *interrupt* which are used to prevent variables being cleared on startup, and to handle interrupts using C code. If this option is used, these keywords, for example, are changed to `__persistent` and `__interrupt` respectively so as to strictly conform to the ANSI standard. This is the same as the `-STRICT` option when compiling from the command line.

Pre-process assembler files

Selecting this item will make HTLPIC pass assembler files through the pre-processor before assembling. This makes it possible to use C pre-processor macros and conditionals in assembler files. A mark appears before the item when it is selected.

Generate assembler listing

This menu option tells the assembler to generate a listing file for each C or assembler source file which is compiled. The name of the list file is determined from the name of the symbol file, for example *led.c* will produce a listing file called *led.lst*.

Generate C source listing

Selecting this option will cause a C source listing for each C file compiled. The listing file will be named in the same way as an assembler file (described above) but will contain the C source code with line numbers, and with tabs expanded. The tab expansion setting is derived from the editor tab stop setting.

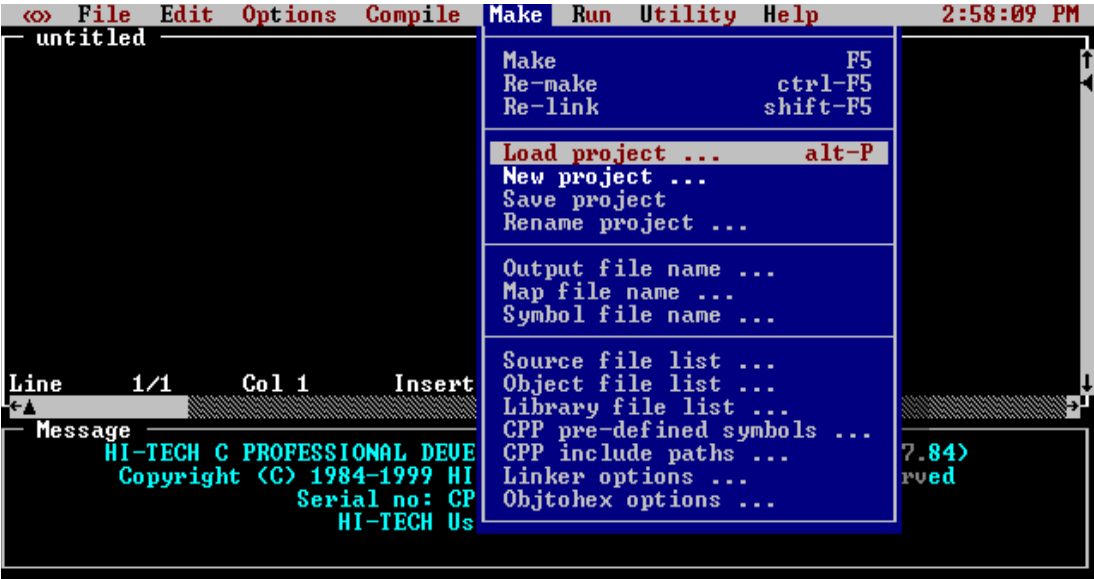
You can only generate *either* a C source listing *or* an assembler listing.

3.5.6 Make menu

The **Make** menu (Figure 3 - 9 on page 82) contains all of the commands required to use the HTLPIC *project* facility. The project facility allows creation of complex multiple-source file applications with ease, as well as a high degree of control of some internal compiler functions and utilities. To use the project facility, it is necessary to follow several steps.

- ☐ Create a new project file using the **New project ...** command. After selecting the project file name, HTLPIC will present several dialogs to allow you to set up options including the processor type and optimisation level.
- ☐ Enter the list of source file names using the **Source file list ...** command.
- ☐ Set up any special libraries, pre-defined pre-processor symbols, object files or linker options using the other items in the **Make** menu.
- ☐ Save the project file using the **Save project** command.
- ☐ Compile your project using the **Make** or **Re-Make** command.

Figure 3 - 9 HTLPIC Make Menu



Make **F5**

The Make command re-compiles the current project. When Make is selected, HTLPIC re-compiles any source files which have been modified since the last Make command was issued. HTLPIC determines whether a source file should be recompiled by testing the modification time and date on the source file and corresponding object file. If the modification time and date on the source file is more recent than that of the object file, it will be re-compiled.

If all object (*.obj*) files are current but the output file cannot be found, HTLPIC will re-link using the object files already present. If all object files are current and the output file is present and up to date, HTLPIC will print a message in the message window indicating that nothing was done.

HTLPIC will also automatically check dependencies, i.e. it will scan source files to determine what files are included, and will include those files in the test to determine if a file needs to be recompiled. In other words, if you modify a header file, any source files including that header file will be recompiled.

If you forget to use the source file list to select the files to be included, HTLPIC will produce a dialog warning that no files have been selected. You will then have to select the DONE button or press *escape*. This takes you back to the editor window.

Re-make**ctrl-F5**

The Re-make command forces recompilation of all source files in the current project. This command is equivalent to deleting all object files and then selecting **Make**.

Re-link**shift-F5**

The Re-link command relinks the current project. Any object files which are missing or not up to date will be regenerated.

Load project ...**alt-P**

This command loads a pre-defined project file. You are presented with a file selection dialog allowing a *.prj* file to be selected and loaded. If this command is selected when the current project has been modified but not saved, you will be given a chance to save the project or abort the **Load project** command. After loading a project file, the message window title will be changed to display the project file name.

New project ...

This command allows the user to start a new project. All current project information is cleared and all items in the Make menu are enabled. The user will be given a chance to save any current project and will then be prompted for the new project's name.

Following entry of the new name HTLPIC will present several dialogs to allow you to configure the project. These dialogs will allow you to select: processor type; float type; output file type; optimisation settings; and map and symbol file options. You will be asked to enter source file names via the *Source file list*.

Save project

This item saves the current project to a file.

Rename project...

This will allow you to specify a new name for the project. The next time the project is saved it will be saved to the new file name. The existing project file will not be affected if it has already been saved.

Output file name ...

This command allows the user to select the name of the compiler output file. This name is automatically setup when a project is created. For example if a project called *prog1* is created and a COD file is being generated, the output file name will be automatically set to *prog1.cod*.

Map file name ...

This command allows the user to enable generation of a symbol map for the current project, and specify the name of the map. The default name of the map file is generated from the project name, e.g. *prog1.map*.

Symbol file name ...

This command allows you to select generation of a symbol file, and specification of the symbol file name. The default name of the symbol file will be generated from the project name, e.g. *progl.sym*. The symbol file produced is suitable for use with MPLAB.

Source file list ...

This option displays a dialog which allows a list of source files to be edited. The source files for the project should be entered into the list, one per line. When finished, the source file list can be exited by pressing *escape*, clicking the mouse on the DONE button, or clicking the mouse in the menu bar.

The source file list can contain any mix of C and assembly language source files. C source files should have the suffix *.c* and assembly language files the suffix *.as*, so that HTLPIC can determine where the files should be passed.

Object file list ...

This option allows any extra *.obj* files to be added to the project. Only enter one *.obj* file per line. Operation of this dialog is the same as the source file list dialog.

This list will normally only contain one object file: the run-time start off module for the current processor. For example, if a project is generating code for the PIC16C84, by default this list will contain a runtime startoff module called *picrt400.obj*. Object files corresponding to files in the source file list SHOULD NOT be entered here as *.obj* files generated from source files are automatically used. This list should only be used for extra *.obj* files for which no source code is available, such as run-time startoff code or utility functions brought in from an outside source.

If a large number of *.obj* files need to be linked in, they should be condensed into a single *.lib* file using the LIBR utility and then accessed using the **Library file list ...** command.

Library file list ...

This command allows any extra object code libraries to be searched when the project is linked. This list normally only contains the default library for the processor being used. For example, if the current project is for a PIC16C84, this list will contain the library *pic400-c.lib*. If an extra library, brought in from an external source, is required, it should be entered here.

It is a good practice to enter any non-standard libraries before the standard C libraries, in case they reference extra standard library routines. The normal order of libraries should be user libraries then the standard C library. Sometimes it is necessary to scan a user library more than once. In this case you should enter the name of the library more than once.

CPP pre-defined symbols ...

This command allows any special pre-defined symbols to be defined. Each line in this list is equivalent to a *-D* option to the command line compiler PCC. For example, if a CPP macro called DEBUG with

value 1, needs to be defined, add the line `DEBUG=1` to this list. Some standard symbols will be pre-defined in this list, these should not be deleted as some of the standard header files rely on their presence.

CPP include paths ...

This option allows extra directories to be searched by the C pre-processor when looking for header files. When a header file enclosed in angle brackets is included, for example `<stdio.h>`, the compiler will search each directory in this list until it finds the file.

Linker options ...

This command allows the options passed to the linker by HTLPIC to be modified. The default contents of the linker command line are generated by the compiler from information selected in the **Options** menu. **You should only use this command if you are sure you know what you are doing!**

Objtohex options ...

This command allows the options passed to objtohex by HTLPIC to be modified. Normally you will not need to change these options as the generation of output files can be chosen in the **Options** menu. However, if you want to generate one of the unusual output formats which objtohex can produce, like COFF files, you will need to change the options using this command.

3.5.7 Run menu

The **Run** menu shown in Figure 3 - 10 on page 86, contains options allowing MS-DOS commands and user programs to be executed.

DOS command ...

alt-d

This option allows an MS-DOS command to be executed exactly like it had been entered at the *command.com* prompt. This command could be an internal MS-DOS command like *dir*, or the name of a program to be executed. If you want to escape to the MS-DOS command processor, use the **DOS Shell** command below.

Warning: do not use this option to load TSR programs.

DOS Shell

alt-J

This item will invoke an MS-DOS *command.com* shell, i.e. you will be immediately presented with a MS-DOS prompt, unlike the **DOS command** item which prompts for a command. To return to HTLPIC, type *exit* at the MS-DOS prompt.

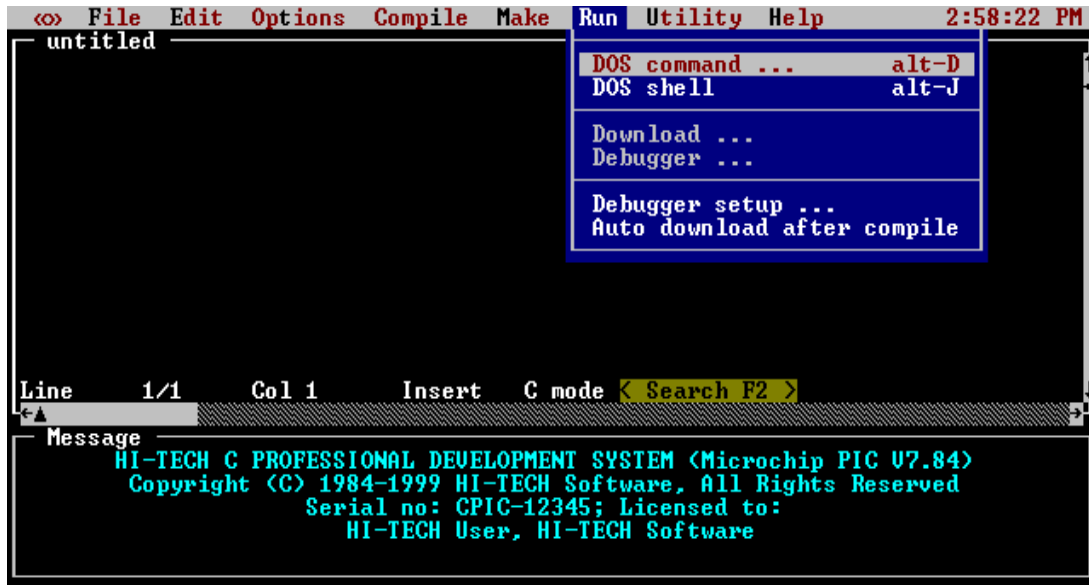
Other Options

All other options in this menu are for future enhancements to the compiler.

3.5.8 Utility menu

The **Utility** menu (Figure 3 - 11 on page 87) contains any useful utilities which have been included in HTLPIC.

Figure 3 - 10 HTLPIC Run Menu

**String search ...**

This option allows you to conduct a string search in a list of files. The option produces a dialog which enables you to type in the string you are seeking and then select a list of files to search. You can also select case sensitivity. It is possible to limit the search to a source file list or just the current project.

Memory usage map

This option displays a window which contains a detailed memory usage map of the last program which was compiled.

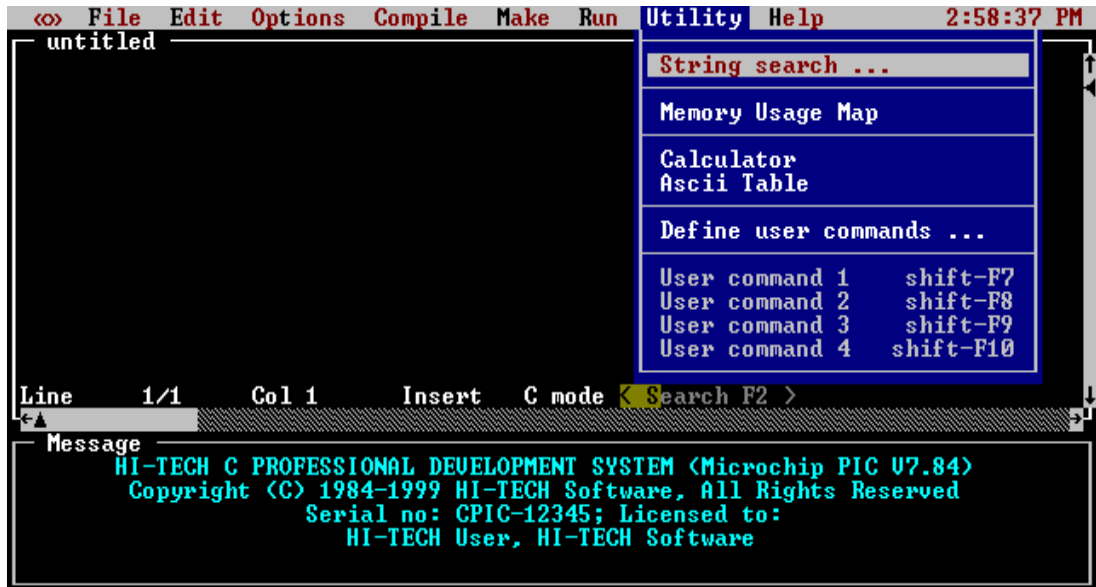
The memory usage map window may be closed by clicking the mouse on the close box in the top left corner of the frame, or by pressing *esc* while the memory map is the front most window.

Calculator

This command selects the HI-TECH Software programmer's calculator. This is a multi-display integer calculator capable of performing calculations in bases 2 (binary), 8 (octal), 10 (decimal) and 16 (hexadecimal). The results of each calculation are displayed in all four bases simultaneously.

Operation is just like a “real” calculator - just press the buttons! If you have a mouse you can click on the buttons on screen, or just use the keyboard. The large buttons to the right of the display allow you to select which radix is used for numeric entry.

Figure 3 - 11 HTLPIC Utility Menu



The calculator window can be moved at will, and thus can be left on screen while the editor is in use. The calculator window may be closed by clicking the OFF button in the bottom right corner, by clicking the close box in the top left corner of the frame, or by pressing *esc* while the calculator is the front most window.

Ascii Table

This option selects a window which contains an ASCII look-up table. The ASCII table window contains four buttons which allow you to close the window or select display of the table in octal, decimal or hexadecimal.

The ASCII table window may be closed by clicking the CLOSE button in the bottom left corner, by clicking the close box in the top left corner of the frame, or by pressing *esc* while the ASCII table is the front most window.

Define user commands...

In the Utility menu are four user-definable commands. This item will invoke a dialog box which will allow you to define those commands. By default the commands are dimmed (not selectable) but will be enabled when a command is defined. Each command is in the form of a DOS command, with macro substitutions available. The macros available are listed in Table 3 - 9 on page 88. Each user-defined command has a hot key associated. They are *shift F7* through *shift F10*, for commands 1 to 4. When a

Table 3 - 9 Macros usable in user commands

Macro name	Meaning
\$(LIB)	Expands to the name of the system library file directory; eg <i>c:\ht-picl\lib\</i>
\$(CWD)	The current working directory
\$(INC)	The name of the system include directory
\$(EDIT)	The name of the file currently loaded into the editor. If the current file has been modified, this will be replaced by the name of the auto saved temporary file. On return this will be reloaded if it has changed.
\$(OUTFILE)	The name of the current output file, i.e. the executable file.
\$(PROJ)	The base name of the current project, eg if the current project file is <i>audio.prj</i> , this macro will expand to <i>audio</i> with no dot or file type.

user command is executed, the current edit file, if changed, will be saved to a temporary file, and the \$(EDIT) macro will reflect the saved temp file name, rather than the original name. On return, if the temp file has changed it will be reloaded into the editor. This allows an external editor to be readily integrated into HTLPIC.

3.5.9 Help menu

The **Help** menu (Figure 3 - 12 on page 89) contains items allowing you to obtain help about any topics listed.

On startup, HTLPIC searches the current directory and the help directory for *tbl* files, which are added to the **Help** menu. The path of the help directory can be specified by the environment variable HT_PICL_HLP. If this is not set, it will be derived from the full path name used when HTLPIC was invoked. If the help directory cannot be located, none of the standard help entries will be available.

HI-TECH Software

This includes information on contacting HI-TECH Software and the licence agreement.

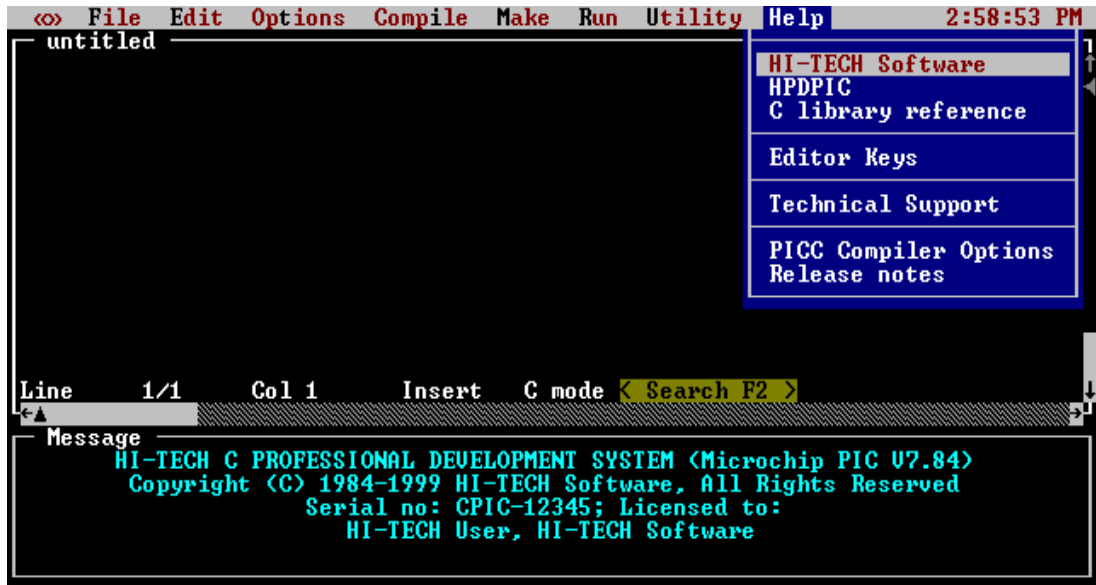
HTLPIC

This option produces a window showing all the topics for which help is available. Topics include Chip types, Compiler optimizations, Editor Searching, Floating point sizes, String search and User defined commands.

C Library Reference

This command selects an on-line manual for the standard ANSI C library. You will be presented with a window containing the index for the manual. Topics can be selected by double clicking the mouse on them, or by moving the cursor with the arrow keys and pressing *return*.

Figure 3 - 12 HTLPIC Help Menu



Once a topic has been selected, the contents of the window will change to an entry for that topic in a separate window. You can move around within the reference using the keypad cursor keys and the index can be re-entered using the INDEX button at the bottom of the window.

If you have a mouse, you can follow *hypertext* links by double clicking the mouse on any word. For example, if you are in the *tan* entry and double click on the reference to *asin*, you will be taken to the entry for *asin*.

This window can be re-sized and moved at will, and thus can be left on screen while the editor is in use.

Editor Keys

This option displays a list editor commands and the corresponding keys used to activate that command.

Technical Support

This option displays a list of dealers and their phone numbers for you to use should you require technical support.

PICL Compiler Options

This option displays a window showing all the PICL compiler options. They are displayed in a table showing the option and its meaning. You can scroll through the table using the normal scroll keys or the mouse.

Release notes

This option displays the release notes for your program. You can scroll through the window using the normal scrolling keys or the mouse.

PICL Command Line Compiler Driver

PICL is invoked from the MS-DOS command line to compile and/or link C programs. If you prefer to use an integrated environment then see the *Using HTLPIC* chapter. PICL has the following basic command format:

```
PICL [options] files [libraries]
```

It is conventional to supply the options (identified by a leading dash ‘-’) before the filenames, but in fact this is not essential.

The options are discussed below. The files may be a mixture of source files (C or assembler) and object files. The order of the files is not important, except that it will affect the order in which code or data appears in memory. The libraries are a list of library names, or *-L* options (see page 99). Source files, object files and library files are distinguished by PICL solely by the file type or extension. Recognized file types are listed in Table 4 - 1. This means, for example, that an assembler file must always have a file type of *.as* (alphabetic case is not important).

Table 4 - 1 PICL File Types

File Type	Meaning
.c	C source file
.as	Assembler source file
.obj	Object code file
.lib	Object library file

PICL will check each file argument and perform appropriate actions. C files will be compiled; assembler files will be assembled. At the end, unless suppressed by one of the options discussed later, all object files resulting from a compilation or assembly, or listed explicitly, will be linked with any specified libraries. Functions in libraries will be linked only if referenced.

Invoking PICL with only object files as arguments (i.e. no source files) will mean only the link stage is performed. It is typical in Makefiles to use PICL with a *-C* option to compile several source files to object files, then to create the final program by invoking PICL with only object files and libraries (and appropriate options).

4.1 Long Command Lines

Since DOS has a command line limitation of 128 characters, to invoke PICL with a long list of options and files, you may create a command file containing the PICL command line, and invoke PICL with its

input redirected from that file. With no command line options specified, PICL will read its standard input to get the argument list. For example a command file may contain:

```
-v -o -16C84 -UBROF -D32
file1.obj file2.obj mylib.lib
```

If this was in the file *xyz.cmd* then PICL would be invoked as:

```
PICL < xyz.cmd
```

Since no command line arguments were supplied, PICL will read *xyz.cmd* for its command line.

In a batch file, a backslash character (\) followed by a return may be used to split a large number of command line options over several lines.

4.2 Default Libraries

PICL will search the standard C library by default. This will always be done last, after any user-specified libraries. The particular library used will be dependent on the processor.

4.3 Standard Run-Time Startoff

PICL will also automatically provide the standard run-time startoff module appropriate. If you require any special powerup initialization, rather than replace or modify the standard run-time startoff module, you should use the *powerup* routine feature (see page 136).

4.4 PICL Compiler Options

The compiler is configured primarily for generation of ROM code. PICL recognizes the compiler options listed in Table 4 - 2 on page 93. The PICL command also allows access to a number of advanced compiler features which are not available within the HTLPIC integrated development environment.

4.4.1 -processor: Define processor

This option defines the processor which is being used. Table 4 - 3 on page 94 shows the processors which are available. An example is; -16C84, to specify the PIC16C84 processor. For more information about this, see Processor Support on page 105.

4.4.2 -A-option: Specify Extra Assembler Option

The -A option can also be used to specify an extra “-” option which will be passed directly to the assembler by PICC. If -A is followed immediately by any text starting with a “-” character, the text will be passed directly to the assembler without being interpreted by PICC. For example, if the option -A-H is specified, the -H option will be passed on to the assembler when it is invoked which will display constant values as hexadecimal values in the assembler output.

Table 4 - 2 PICL Options

Option	Meaning
<i>-processor</i>	Define the processor
<i>-A-option</i>	Specify <i>-option</i> to be passed directly to the assembler
<i>-AAHEX</i>	Generate an American Automation symbolic HEX file
<i>-ASMLIST</i>	Generate assembler .LST file for each compilation
<i>-BIN</i>	Generate a Binary output file
<i>-C</i>	Compile to object files only
<i>-CKfile</i>	Make OBJTOHEX use a checksum file
<i>-CRfile</i>	Generate cross-reference listing
<i>-D24</i>	Use truncated 24-bit floating point format for doubles
<i>-D32</i>	Use IEEE754 32-bit floating point format for doubles
<i>-Dmacro</i>	Define pre-processor macro
<i>-E</i>	Define format for compiler errors
<i>-Efile</i>	Redirect compiler errors to a file
<i>-E+file</i>	Append errors to a file
<i>-FAKELOCAL</i>	Produce MPLAB-specific debug information
<i>-FDOUBLE</i>	Enables the use of faster 32-bit floating point math routines.
<i>-Gfile</i>	Generate enhanced source level symbol table
<i>-HELP</i>	Print summary of options
<i>-Ipath</i>	Specify a directory pathname for include files
<i>-INTEL</i>	Generate an Intel HEX format output file (default)
<i>-Llibrary</i>	Specify a library to be scanned by the linker
<i>-L-option</i>	Specify <i>-option</i> to be passed directly to the linker
<i>-Mfile</i>	Request generation of a MAP file
<i>-MOT</i>	Generate a Motorola S1/S9 HEX format output file
<i>-Nsize</i>	Specify identifier length
<i>-NORT</i>	Do not link standard runtime module
<i>-O</i>	Enable post-pass optimization
<i>-Ofile</i>	Specify output filename
<i>-P</i>	Preprocess assembler files
<i>-PRE</i>	Produce preprocessed source files
<i>-PROTO</i>	Generate function prototype information
<i>-PSECTMAP</i>	Display complete memory segment usage after linking
<i>-q</i>	Specify quiet mode
<i>-RESRAM=ranges</i>	Reserve the specified RAM address ranges.
<i>-RESROM=ranges</i>	Reserve the specified ROM address ranges.
<i>-ROMranges</i>	Specify external ROM memory ranges available
<i>-S</i>	Compile to assembler source files only

Table 4 - 2 PICL Options

Option	Meaning
-SIGNED_CHAR	Make the default char signed.
-STRICT	Enable strict ANSI keyword conformance
-TEK	Generate a Tektronix HEX format output file
-U <i>symbol</i>	Undefine a predefined pre-processor symbol
-UBROF	Generate an UBROF format output file
-V	Verbose: display compiler pass command lines
-W <i>level</i>	Set compiler warning level
-X	Eliminate local symbols from symbol table
-Zg[<i>level</i>]	Enable global optimization in the code generator

4.4.3 -AAHEX: Generate American Automation Symbolic Hex

The -AAHEX option directs PICL to generate an American Automation symbolic format HEX file, producing a file with the *.hex* extension. This option has no effect if used with a *.bin* file. The American Automation hex format is an enhanced Motorola S-Record format which includes symbol records at the start of the file. This option should be used if producing code which is to be debugged with an American Automation in-circuit emulator.

Table 4 - 3 Processors

Midrange Processors
PIC16C84
PIC16F84
PIC16F84A
PIC16F627

4.4.4 -ASMLIST: Generate Assembler .LST Files

The -ASMLIST option tells PICL to generate an assembler *.LST* file for each compilation. The list file shows both the original C code, and the generated assembler code and the corresponding binary code. The listing file will have the same name as the source file, and a file type (extension) of *.lst*.

4.4.5 -BIN: Generate Binary Output File

The -BIN option tells PICL to generate a Binary image output file. The output file will be given type *.bin*. Binary output may also be selected by specifying an output file of type *.bin* using the -O*file* option.

4.4.6 -C: Compile to Object File

The `-C` option is used to halt compilation after generating an object file. This option is frequently used when compiling multiple source files using a “make” utility. If multiple source files are specified to the compiler each will be compiled to a separate *.obj* file. To compile three source files *main.c*, *module1.c* and *asmcode.as* to object files you could use the command:

```
PICL -16C84 -O -Zg -C main.c module1.c asmcode.as
```

The compiler will produce three object files *main.obj*, *module1.obj* and *asmcode.obj* which could then be linked to produce a Motorola HEX file using the command:

```
PICL -16C84 main.obj module1.obj asmcode.obj
```

The compiler will accept any combination of *.c*, *.as* and *.obj* files on the command line. Assembler source files will be passed directly to the assembler and object files will not be used until the linker is invoked. Unless the `-Ofile` option is used to specify an output file name and type the final output will be a Motorola hex file with the same “base name” as the first source or object file, the example above would produce a file called *main.hex*.

4.4.7 -CKfile: Generate Check Sum

This option causes OBJTOHEX to use *file* for checksum specifications. See Objtohex section for further details.

4.4.8 -CRfile: Generate Cross Reference Listing

The `-CR` option will produce a cross reference listing. If the *file* argument is omitted, the “raw” cross reference information will be left in a temporary file, leaving the user to run the CREF utility. If a filename is supplied, for example `-CRtest.crf`, PICL will invoke CREF to process the cross reference information into the listing file, in this case *TEST.CRF*. If multiple source files are to be included in the cross reference listing, all must be compiled and linked with the one PICL command. For example, to generate a cross reference listing which includes the source modules *main.c*, *module1.c* and *nvr.am.c*, compile and link using the command:

```
PICL -16C84 -CRmain.crf main.c module1.c nvr.am.c
```

4.4.9 -D24: Use 24-bit Doubles

This option is the default, causing the use of truncated 24-bit floating point format for doubles. See Floating Point on page 114 for more details.

4.4.10 -D32: Use 32-bit Doubles

This tells the compiler to use the IEEE754 32-bit floating point format for doubles. See Floating Point on page 114 for more details.

4.4.11 -Dmacro: Define Macro

The `-D` option is used to define a preprocessor macro on the command line, exactly as if it had been defined using a `#define` directive in the source code. This option may take one of two forms, **-Dmacro** which is equivalent to:

```
#define macro 1
```

or **-Dmacro=text** which is equivalent to:

```
#define macro text
```

Thus, the command:

```
PICL -16C84 -Ddebug -Dbuffers=10 test.c
```

will compile `test.c` with macros defined exactly as if the C source code had included the directives:

```
#define debug 1
#define buffers 10
```

4.4.12 -E: Define Format for Compiler Errors

If the `-E` option is not used, the default behaviour is to display compiler errors in a “human readable” format line with a caret and error message pointing out the offending characters in the source line, for example:

```
x.c: main()
      4: PORT_A = xFF;
                ^ undefined identifier: xFF
```

The standard format is perfectly acceptable to a person reading the error output but is not usable with editors which support compiler error handling.

4.4.12.1 Using the -E Option

Using the `-E` option instructs the compiler to generate error messages in a format which is acceptable to some text editors.

If the same source code as used in the example above were compiled using the `-E` option, the error output would be:

```
x.c 4 9: undefined identifier: xFF
```

indicating that the error occurred in file `x.c` at line 4, offset 9 characters into the statement. The second numeric value, the column number, is relative to the left-most non-space character on the source line. If

an extra space or tab character were inserted at the start of the source line, the compiler would still report an error at line 4, column 9.

4.4.12.2 Modifying the Standard -E Format

If the -E option does not meet your editor's requirement, you can redefine its format by setting two environment variables: HTC_ERR_FORMAT and HTC_WARN_FORMAT. These environment variables are in the form of a *printf*-style string in which you can use the specifiers shown in Table 4 - 4.

Table 4 - 4 Error Format Specifiers

Specifier	Expands To
%f	Filename
%l	Line number
%c	Column number
%s	Error string

The column number is relative to the left-most non-space character on the source line. Here is an example of setting the environment variables:

```
set HTC_WARN_FORMAT=WARNING: file %f; line %l; column %c; %s
set HTC_ERR_FORMAT=ERROR: file %f; line %l; column %c; %s
```

Using the previous source code, the output from the compiler when using the above environment variables would be:

```
ERROR: file x.c; line 4; column 9; undefined identifier: xFF
```

Remember that if these environment variables are set in a batch file, you must prepend the specifiers with an additional percent character to stop the specifiers being interpreted immediately by DOS, e.g. %%f.

4.4.12.3 Redirecting Errors to a File

Error output, either in standard or -E format, can be redirected into files using UNIX or MS-DOS style standard output redirection. The error from the example above could have been redirected into a file called *errlist* using the command:

```
PICL -16C84 -E x.c > errlist
```

Compiler errors can also be appended onto existing files using the redirect and append syntax. If the error file specified does not exist it will be created. To append compiler errors onto a file use a command like:

```
PICL -16C84 -E x.c >> errlist
```

4.4.13 -Efile: Redirect Compiler Errors to a File

Some editors do not allow the standard command line redirection facilities to be used when invoking the compiler. To work with these editors, PICL allows the error listing file name to be specified as part of the -E option. Error files generated using this option will always be in -E format. For example, to compile `x.c` and redirect all errors to `x.err`, use the command:

```
PICL -16C84 -Ex.err x.c
```

The -E option also allows errors to be appended to an existing file by specifying a + at the start of the error file name, for example:

```
PICL -16C84 -E+x.err y.c
```

If you wish to compile several files and combine all of the errors generated into a single text file, use the -E option to create the file then use -E+ when compiling all the other source files. For example, to compile a number of files with all errors combined into a file called `project.err`, you could use the -E option as follows:

```
PICL -16C84 -Eproject.err -O -Zg -C main.c
PICL -16C84 -E+project.err -O -Zg -C part1.c
PICL -16C84 -E+project.err -C asmcode.as
```

The file `project.err` will contain any errors from `main.c`, followed by the errors from `part1.c` and then `asmcode.as`, for example:

```
main.c 11 22: ) expected
main.c 63 0: ; expected
part1.c 5 0: type redeclared
part1.c 5 0: argument list conflicts with prototype
asmcode.as 14 0: Syntax error
asmcode.as 355 0: Undefined symbol _putint
```

4.4.14 -FDOUBLE

This option is used to enable the use of faster 32-bit floating point routines. These alternative routines make floating point multiplication and division many times faster than the default routines, but do require more ROM and RAM space. These routines are only available for Highend processors and for 32-bit double types.

4.4.15 -FAKELOCAL

This option should be used in conjunction with the -G option to produce debug information that is specific to MPLAB. It will allow MPLAB to examine local variables. Information associated with source-level single stepping is also modified. See also Section 5.36 on page 142.

4.4.16 -Gfile: Generate Source Level Symbol File

-G generates a source level symbol file for use with HI-TECH Software debuggers and simulators such as *Lucifer*. If no filename is given, the symbol file will have the same “base name” as the first source or object file, and an extension of `.SYM`. For example, `-GTEST.SYM` generates a symbol file called `TEST.SYM`. Symbol files generated using the -G option include source level information for use with source level debuggers.

Note that all source files for which source level debugging is required should be compiled with the -G option. The option is also required at the link stage, if this is performed separately. For example:

```
PICL -16C84 -G -C test.c
PICL -16C84 -C module1.c
PICL -16C84 -Gtest.sym test.obj module1.obj
```

will include source level debugging information for `test.c` only because `module1.c` was not compiled with the -G option.

4.4.17 -HELP: Display Help

When used with no other options present on the command line, the -HELP option displays information on the PICL options.

4.4.18 -Ipath: Include Search Path

Use -I to specify an additional directory to use when searching for header files which have been included using the `#include` directive. The -I option can be used more than once if multiple directories are to be searched. The default include directory containing all standard header files will still be searched, after any user specified directories have been searched. For example:

```
PICL -16C84 -C -Ic:\include -Id:\myapp\include test.c
```

will search the directories `c:\include` and `d:\myapp\include` for any header files included using angle brackets.

4.4.19 -INTEL: Generate INTEL Hex File

The -INTEL option directs PICL to generate an Intel HEX file if producing a file with `.HEX` extension. This option has no effect if used with a `.BIN` file.

4.4.20 -Library: Scan Library

The -L option is used to specify additional libraries which are to be scanned by the linker. Libraries specified using the -L option are scanned before the standard C library, allowing additional versions of standard library functions to be accessed.

The argument to `-L` is a library keyword to which the prefix `PIC`; numbers representing the processor range, number of ROM banks and the number of RAM banks; and suffix `.LIB` are added. Thus the option `-LL` will, for example, scan the library `PIC411-L.LIB` and the option `-LXX` will scan a library called `PIC411-XX.LIB`. All libraries must be located in the `LIB` subdirectory of the compiler installation directory.

If you wish the linker to scan libraries whose names do not follow this naming convention or whose locations are not in the `LIB` subdirectory, include the libraries' names on the command line along with your source files. Alternatively, the additional libraries can be specified in the HTLPIC **library file list** menu, or the linker may be invoked directly.

The complete set of libraries and runtime startoff modules supplied with the compiler is listed in Table 5 - 1 on page 106.

4.4.21 `-L-option`: Specify Extra Linker Option

The `-L` option can also be used to specify an extra “-” option which will be passed directly to the linker by PICL. If `-L` is followed immediately by any text starting with a “-” character, the text will be passed directly to the linker without being interpreted by PICL. For example, if the option `-L-FOO` is specified, the `-FOO` option will be passed on to the linker when it is invoked. The `-L` option is especially useful when linking code which contains extra program sections (or *psects*, as may be the case if the program contains assembler code or C code which makes use of the `#pragma psect` directive. If the `-L` option did not exist, it would be necessary to invoke the linker manually or use an HTLPIC option to link code which uses extra psects. The `-L` option makes it possible to specify any extra psects simply by using an extra linker `-P` option. To give a practical example, suppose your code contains variables which have been mapped into a special RAM area using an extra psect called *xtraram*. In order to link this new psect at the appropriate address all you need to do is pass an extra linker `-P` option using the `-L` option. For example, if the special RAM area (*xtraram* psect) were to reside at address 50h, you could use the PICL option `-L-Pxtreram=50h` as follows:

```
PICL -16C84 -L-Pxtraram=50h prog.c xram.c
```

One commonly used linker option is `-N`, which sorts the symbol table in the map file in address rather than name order. This is passed to PICL as `-L-N`.

4.4.22 `-Mfile`: Generate Map File

The `-M` option is used to request the generation of a map file. If no filename is specified, the map information is displayed on the screen, otherwise the filename specified to `-M` will be used.

4.4.23 `-MOT`: Generate Motorola S-Record HEX File

The `-MOT` option directs PICL to generate a Motorola S-Record HEX file if producing a file with `.HEX` extension. This option has no effect if used with a `.BIN` file.

4.4.24 -Nsize: Identifier Length

This option allows the C identifier length to be increased from the default value of 31. Valid sizes are from 32 to 255. The option has no effect for all other values.

4.4.25 -NORT: Do Not Link Standard Runtime Module

Using this option will not link in the standard runtime startup module. The user should then supply their own version of the runtime startup module in the list of input files on the command line. Even if the required startup module does not contain executable code, it will almost certainly require symbol and psect definitions for successful compilation, so this module cannot simply be omitted completely. The source for the standard runtime module is supplied in the SOURCES directory of your distribution and this should be used as the basis for your own runtime module.

4.4.26 -O: Invoke Optimizer

-O invokes the post-pass optimizer after the code generation pass.

4.4.27 -Ofile: Specify Output File

This option allows the name and type of the output file to be specified to the compiler. If no -O option is specified, the output file will be named after the first source or object file. You can use the -O option to specify an output file of type HEX, BIN or UBR, containing HEX, Binary or UBROF respectively. For example:

```
PICL -16C84 -Otest.bin prog1.c part2.c
```

will produce a binary file named *test.bin*.

4.4.28 -P: Pre-process Assembly Files

-P causes the assembler files to be pre-processed before they are assembled.

4.4.29 -PRE: Produce Pre-processed Source Code

-PRE is used to generate pre-processed C source files with an extension .PRE. It may be useful to ensure that macros expand to what you think they should. Use of this option can also create C source files which do not require any separate header files. This is useful when sending files for technical support.

4.4.30 -PROTO: Generate Prototypes

-PROTO is used to generate .PRO files containing both ANSI and K&R style function declarations for all functions within the specified source files. Each .PRO file produced will have the same base name as the corresponding source file. Prototype files contain both ANSI C style prototypes and old style C function declarations within conditional compilation blocks.

The *extern* declarations from each .PRO file should be edited into a global header file which is included in all the source files comprising a project. The .PRO files may also contain *static* declarations for functions which are local to a source file. These *static* declarations should be edited into the start of the source file. To demonstrate the operation of the -PROTO option, enter the following source code as file *test.c*:

```
#include <stdio.h>
add(arg1, arg2)
int *   arg1;
int *   arg2;
{
    return *arg1 + *arg2;
}

void printlist(int * list, int count)
{
    while (count--)
        printf("%d ", *list++);
    putchar('\n');
}
```

If compiled with the command `PICL -16C84 -PROTO test.c`, PICL will produce *test.pro* containing the following declarations which may then be edited as necessary:

```
/* Prototypes from test.c */
/* extern functions - include these in a header file */
#if     PROTOTYPES
extern int add(int *, int *);
extern void printlist(int *, int);
#else /* PROTOTYPES */
extern int add();
extern void printlist();
#endif /* PROTOTYPES */
```

4.4.31 -PSECTMAP: Display Complete Memory Usage

The -PSECTMAP option is used to display a complete memory and psect (*program section*) dump after linking the user code. The information provided by this option is more detailed than the standard memory usage map which is normally printed after linking. The -PSECTMAP option causes the compiler to print a listing of every compiler and user generated psect, followed by the standard memory usage map. For example:

Psect Usage Map:

Psect	Contents	Memory Range
-----	-----	-----
powerup	Power on reset code	\$0000 - \$0003
init	Initialization code	\$0004 - \$0007
end_init	Initialization code	\$0008 - \$000B
clrtext	Memory clearing code	\$000C - \$0012
text	Program and library code	\$0745 - \$074C
text1	Program and library code	\$074D - \$075D
ftext	Arithmetic routine code	\$075E - \$0769
float_te	Arithmetic routine code	\$076A - \$07FF
rbss_0	Bank 0 RAM variables	\$0020 - \$0022
temp	Temporary RAM data	\$0070 - \$007B

Memory Usage Map:

Program ROM	\$0000 - \$0012	\$0013 (19) words
Program ROM	\$0745 - \$07FF	\$00BB (187) words
		\$00CE (206) words total Program ROM
Bank 0 RAM	\$0020 - \$0022	\$0003 (3) bytes
Bank 0 RAM	\$0070 - \$007B	\$000C (12) bytes
		\$000F (15) bytes total Bank 0 RAM

4.4.32 -q: Quiet Mode

If used, this option must be the *first* option. It places the compiler in quiet mode which suppresses the HI-TECH Software copyright notice from being output.

4.4.33 -RESRAMranges[,ranges]

The -RESRAM option is used to reserve a particular section of RAM space. The address ranges must be specified in HEX. The syntax for this option is a comma separated list of address ranges. For example:

```
-RESRAM20-40
```

This will reserve the RAM addresse range from 0x20 to 0x40.

4.4.34 -RESROMranges[,ranges]

The -RESROM option is used to reserve a particular section of ROM space. The address ranges must be specified in HEX. The syntax for this option is a comma separated list of address ranges. For example:

`-RESROM1000-10FF,2000-20FF`

This will reserve the ROM address ranges 0x1000 to 0x10FF and 0x2000 to 0x20FF.

4.4.35 -ROMranges

If external ROM space is available, code can be allocated into these areas with this option. The syntax for this option is a comma separated list of address ranges. For example:

`-ROM1000-1FFF,2000-2FFFF` If external ROM is used, the compiler will make use of the `fcall` and `ljmp` instructions to access it. For more information about `fcall` and `ljmp`, see "Additional Mnemonics" on page 182.

4.4.36 -S: Compile to Assembler Code

The `-S` option stops compilation after generating an assembler source file. An assembler file will be generated for each C source file passed on the command line. The command:

`PICC -16C84 -O -Zg -S test.c`

will produce an assembler source file called `test.as` which contains the code generated from `test.c`. The optimization options `-O` and `-Zg` can be used with `-S`, making it possible to examine the compiler output for any given set of options. This option is particularly useful for checking function calling conventions and "signature" values when attempting to write external assembly language routines.

4.4.37 -SIGNED_CHAR: Make Char Type Signed

Unless this option is used, the default behaviour of the compiler is to make all character values and variables *unsigned char* unless explicitly declared or cast to *signed char*. This option will make the default char type *signed char*. Any unsigned char will have to be explicitly declared *unsigned char*.

The range of *signed char* is -128 to +127 and the range of *unsigned char* is 0 to 255

4.4.38 -STRICT: Strict ANSI Conformance

The `-STRICT` option is used to enable strict ANSI conformance of all special keywords. HI-TECH C supports various special keywords (for example *bank1* type qualifier). If the `-STRICT` option is used, these keywords are changed to include a double underscore at the beginning (e.g. `__bank1`) so as to strictly conform to the ANSI standard. Be warned that use of this option may cause problems with some standard header files (e.g. *intrpt.h*).

4.4.39 -TEK: Generate Tektronix HEX File

The `-TEK` option tells the compiler to generate a Tektronix format HEX file if producing a file with `.HEX` extension. This option has no effect if used with a `.BIN` file.

4.4.40 -Umacro: Undefine a Macro

-U, the inverse of the -D option, is used to undefine predefined macros. This option takes the form -U*macro*. For example, to remove the pre-defined macro *debug* use the option -U*debug*.

4.4.41 -UBROF: Generate UBROF Format Output File

The -UBROF option tells the compiler to generate a UBROF format output file suitable for use with certain in-circuit emulators. The output file will be given an extension .UBR. UBROF output may also be selected by specifying an output file of type .UBR using the -O option. This option has no effect if used with a .BIN file.

4.4.42 -V: Verbose Compile

-V is the “verbose” option. The compiler will display the command lines used to invoke each of the compiler passes. This option may be useful for determining the exact linker options which should be used if you want to directly invoke the HLINK command.

4.4.43 -Wlevel: Set Warning Level

-W is used to set the compiler warning level. Allowable warning levels range from -9 to 9. The warning level determines how picky the compiler is about dubious type conversions and constructs. The default warning level -W0 will allow all normal warning messages. Warning level -W1 will suppress the message “Func() declared implicit int”. -W3 is recommended for compiling code originally written with other, less strict, compilers. -W9 will suppress all warning messages. Negative warning levels -W-1, -W-2 and -W-3 enable special warning messages including compile-time checking of arguments to *printf()* against the format string specified.

4.4.44 -X: Strip Local Symbols

The option -X strips local symbols from any files compiled, assembled or linked. Only global symbols will remain in any object files or symbol files produced.

4.4.45 -Zg[level]: Global Optimization

The -Zg option invokes global optimization during the code generation pass. This can result in significant reductions to code size and internal RAM usage. This optimizer is less critical than the post-pass optimizer, but can still significantly reduce the code size.

Global optimization attempts to Optimize register usage on a function-by-function basis. It also takes advantage of constant propagation in code to avoid un-necessary accesses to memory.

The default level for this option is 1 (the least optimization). The level can be set anywhere from 1 to 9 (the most optimization). The number indicates how hard the optimizer tries to reduce code size. For PICL, there is usually little advantage in using levels above 3.

4

Features and Runtime Environment

HI-TECH C supports a number of special features and extensions to the C language which are designed to ease the task of producing ROM-based applications. This chapter documents the compiler options and special features which are available. After reading and understanding this manual you should know how to:

- ☐ configure the console I/O routines so that you can use <stdio.h> routines on your hardware.
- ☐ set up the interrupt handler using only C code.
- ☐ program I/O devices using only C code.
- ☐ interface between C and assembler code using inline or external assembly language routines.

5.1 Divergence from the ANSI C Standard

PIC C diverges from the ANSI C standard in one area: function recursion.

Due to the PIC's hardware limitations of no stack and limited memory, function recursion is unsupported.

5.2 Processor Support

PIC C supports a selected range of processors as shown in Table 4 - 3 on page 94.

5.3 Standard Libraries

PIC C includes a number of standard libraries

Figure 5 - 1 on page 106 illustrates the naming convention used for the standard libraries. The meaning of each field is described here, where:

- ☐ *Processor Type* is always *pic*.
- ☐ *Processor Range* is 2 for the Baseline range, 4 for the Midrange and 6 for the High-End range of PIC microprocessors.
- ☐ *# of ROM Banks* is 2^n .
- ☐ *# of RAM Banks* is 2^m .
- ☐ *Double Type* is - for 24-bit doubles, and *d* for 32-bit doubles.

Table 5 - 1 on page 106 lists the standard libraries which are supplied.

Figure 5 - 1 PIC Standard Library Naming Convention

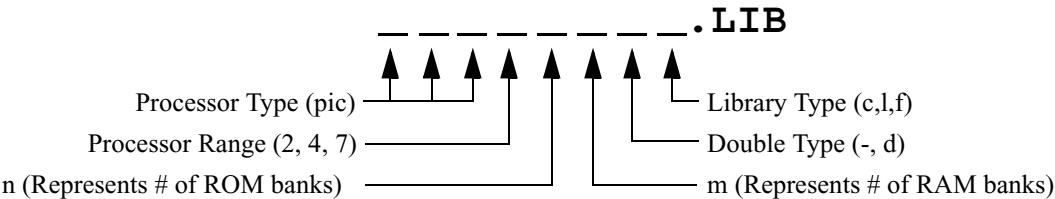


Table 5 - 1 Standard Libraries

Library	Purpose
pic400-c.lib	Midrange processor, 1 ROM bank, 1 RAM banks, 24-bit doubles
pic400dc.lib	Midrange processor, 1 ROM bank, 1 RAM bank, 32-bit doubles

5.4 Output File Formats

The compiler is able to directly produce a number of the output file formats which are used by common PROM programmers and in-circuit emulators.

If you are using the HTLPIC integrated environment compiler driver you can select Motorola Hex, Intel Hex, Binary, UBROF, Tektronix Hex, American Automation symbolic Hex, Bytecraft .COD, or Library file using the **Output file type** menu item in the “Options” menu.

The default behaviour of the PICL command is to produce Bytecraft COD and Intel HEX output. If no output file name or type is specified, PICL will produce a Bytecraft COD and Intel HEX file with the same base name as the first source or object file. Table 5 - 2 on page 107 shows the output format options available with PICL. With any of the output format options, the base name of the output file will be the same as the first source or object file passed to PICL. The *File Type* column lists the filename extension which will be used for the output file.

In addition to the options shown, the **-O** option may be used to request generation of binary or UBROF files. If you use the **-O** option to specify an output file name with a *.BIN* type, for example **-Otest.bin**, PICL will produce a binary file. Likewise, if you need to produce UBROF files, you can use the **-O** option to specify an output file with type *.UBR*, for example **-Otest.ubr**.

5.5 Symbol Files

The PICL **-G** option tell the compiler to produce a symbol file which can be used by debuggers and simulators to perform symbolic and source level debugging. This option produces symbol files which contain assembler level information and C source level information. If no symbol file name is specified, by default a file called *file.sym* will be produced, where *file* is the basename of the first source file on

Table 5 - 2 Output File Formats

Format Name	Description	PICL Option	File Type
Motorola HEX	S1/S9 type hex file	-MOT	.HEX
Intel HEX	Intel style hex records (default)	-INTEL	.HEX
Binary	Simple binary image	-BIN	.BIN
UBROF	“Universal Binary Image Relocatable Format”	-UBROF	.UBR
Tektronix HEX	Tektronix style hex records	-TEK	.HEX
American Auto- mation HEX	Hex format with symbols for American Auto- mation emulators	-AAHEX	.HEX
Bytecraft .COD	Bytecraft code format (default)	n/a (default)	.COD
Library	HI-TECH library file	n/a	.LIB

the command line. For example, to produce a symbol file called *test.sym* which includes C source level information:

```
PICL -16C84 -Gtest.sym test.c
```

The symbol files produced by these options may be used with in-circuit emulators.

5.6 Predefined Macros

The compiler drivers define certain symbols to the preprocessor (CPP), allowing conditional compilation based on chip type, memory model etc. The symbols listed in Table 5 - 3 show the more common symbols defined by the drivers. Each symbol, if defined, is equated to 1 unless otherwise stated.

5.7 Configuration Fuses

The PIC processor’s configuration fuses may be set using the `__CONFIG` macro as follows:

```
#include <pic.h>
__CONFIG(x) ;
```

where x is the word that is to be the configuration word. For convenience, the commonly-used bits are provided in the header file for the appropriate PIC processor. The code protection fuses are not always defined in the header file. Here is an example for the PIC16C5x:

```
__CONFIG(FOSC1|WDTE|0x0600) ;
```

Some PIC processors have more than a single bit location that need to be programmed to disable code space protection. There are special macros defined in the specific PIC header files which may be used to select the available levels of code protection. Check the appropriate header file and ensure that all the

Table 5 - 3 Predefined CPP Symbols

Symbol	When set	Usage
HI_TECH_C	Always	To indicate that the compiler in use is HI-TECH C.
MPC	Always	To indicate the code is compiled for the Microchip PIC family.
_PIC12	If 12-bit device	To indicate that this a Baseline PIC device.
_PIC14	If 14-bit device	To indicate that this a Midrange PIC device.
_PIC16	If 16-bit device	To indicate that this a High-End PIC device.
COMMON	If common RAM present	To indicate the presence of a common RAM area.
BANKBITS	To 0, 1 or 2	Assigned 0, 1 or 2 to indicate 1, 2 or 4 available banks of RAM.
MPLAB_ICD	If using ICD	To indicate that code is being generated for the MPLAB In-Circuit Debugger

configuration bits are correctly specified in the `__CONFIG` macro for your application before programming the device.

5

5.8 ID Locations

Some PIC devices have locations outside the addressable memory area that can be used for storing program information, such as an ID number. The `__IDLOC` macro may be used to place data into these locations. The macro is used in a manner similar to:

```
#include <pic.h>

__IDLOC(x) ;
```

where x is a list of nibbles which are to be positioned in to the ID locations. Only the lower four bits of each ID location is programmed, so the following:

```
__IDLOC(15F0) ;
```

will attempt to fill four ID locations with the decimal values: 1, 5, 15 and 0. The base address of the ID locations is specified by the `idloc` psect which will be automatically assigned an address dependent on the type of processor selected.

5.9 Bit Instructions

Wherever possible, HI-TECH C will attempt to use the PIC bit instructions. For example, when using a bitwise operator and a mask to alter a bit within an integral type, the compiler will check the mask value to determine if a bit instruction can achieve the same functionality.

```
int foo;
foo |= 0x40;
```

will produce the instruction

```
bsf _foo,6
```

To set or clear individual bits within integral types, the following macros could be used.

```
#define bitset(var,bitno) ((var) |= 1 << (bitno))
#define bitclr(var,bitno) ((var) &= ~(1 << (bitno)))
```

To perform the same operation as above, the bitset macro could be employed as follows.

```
bitset(foo,6);
```

5.9.1 The OPTION instruction

Some baseline PIC devices use an *option* instruction to load the option register. The appropriate header files contain a special definition for a C object called `OPTION` and macros for the bit symbols which are stored in this register. HI-TECH C will automatically use the option instruction when an appropriate processor is selected and the `OPTION` object is accessed.

For example, to set the prescaler assignment bit so that prescaler is assigned to the watch dog timer, the following code can be used after including `pic.h`.

```
OPTION = PSA;
```

This will load the appropriate value into the W register and then call the option instruction.

5.9.2 The TRIS instructions

Some PIC devices use a *tris* instruction to load the *tris* register. The appropriate header files contain a special definition for a C object called `TRIS`. HI-TECH C will automatically use the *tris* instruction when an appropriate processor is selected and the `TRIS` object is accessed.

For example, to make all the bits on the output port high impedance, the following code can be used after including `pic.h`.

```
TRIS = 0xFF;
```

This will load the appropriate value into the W register and then call the tris instruction.

Those PIC devices which have more than one output port may have definitions for objects: `TRISA`, `TRISB` and `TRISC`, depending on the exact number of ports available. These objects are used in the same manner as described above.

The byte parameters may be accessed directly using the identifiers defined in the header file.

5.10 Oscillator calibration constants

Some PIC devices come with an oscillator calibration constant which is pre-programmed into the device's program memory. This constant can be read and written to the OSCCAL register to calibrate the internal RC oscillator. On some baseline PIC devices the calibration constant is stored as a RETLW instruction at the top of program memory, e.g. the 12C50X and 16C505 parts. On reset the program counter is made to point to this instruction and it is executed first before the program counter wraps around to 0x0000 which is the "effective" reset vector for the device. The PICL compiler's default startup routine will automatically include code to load the OSCCAL register with the value contained in the W register after reset on such devices. No other code is required by the programmer.

For 12C67X chips the oscillator constant is also stored at the top of program memory, but is not automatically loaded after reset. It can be read at any time during program execution using the macro `_READ_OSCCAL_DATA()`. To be able to use this macro, make sure that `<pic.h>` is included into the relevant modules of your program. This macro returns the calibration constant which should then be stored into the OSCCAL register, as follows:

```
OSCCAL = _READ_OSCCAL_DATA();
```

5

The location which stores the calibration constant is never code protected and will be lost if you reprogram the device. Thus, if you are using a windowed device, the calibration constant must be saved from the last ROM location before it is erased. The constant must then be reprogrammed at the same location along with the new program and data.

If you are using an in-circuit emulator (ICE), the location used by the calibration RETLW instruction may not be programmed and would be executed as some other instruction. Calling the `_READ_OSCCAL_DATA()` macro will not work and will almost certainly not return correctly. If you wish to test code that includes this macro on an ICE, you will have to program a RETLW instruction at the appropriate location in program memory. Remember to remove this instruction when programming the actual part so you do not destroy the calibration value.

5.11 Supported Data Types

The PICL compiler supports basic data types of 1, 2 and 4 byte size. All multi-byte types follow *least significant byte first* format, also known as *little-endian*. Word size values thus have the least significant byte at the lower address, and double word size values have the least significant byte and least significant word at the lowest address.

Table 5 - 4 shows the data types and their corresponding size and arithmetic type.

Table 5 - 4 Data Types

Type	Size (in bits)	Arithmetic Type
bit	1	boolean
char	8	signed or unsigned integer ^a
unsigned char	8	unsigned integer
short	16	signed integer
unsigned short	16	unsigned integer
int	16	signed integer
unsigned int	16	unsigned integer
long	32	signed integer
unsigned long	32	unsigned integer
float	24	real
double	24 or 32 ^b	real

a.A char is unsigned by default, and signed if the PICL
-SIGNED_CHAR option is used.
b.A double defaults to 24-bit, but becomes 32-bit with the PICL
-D32 option.

5.11.1 Radix Specifiers

PICL supports the ANSI standard radix specifiers as well as one which enables binary constants to specified in C code. The format used to specify the radices are given in Table 5 - 5 on page 111. The letters used to specify binary or hexadecimal radices are case insensitive, as are the letters used to specify a hexadecimal digit.

Table 5 - 5 Radix Formats

Radix	Format	Example
binary	0bnumber or 0Bnumber	0b10011010
octal	0number	0763
decimal	number	129
hexadecimal	0xnumber or 0Xnumber	0x2F

5.11.2 Bit Data Types

HI-TECH C allows single bit variables to be declared using the keyword *bit*. A variable declared *bit*, for example:

```
static bit init_flag;
```

will be allocated in the bit addressable psects *rbit_n* (where *n* is the bank number), and will be visible only in that module or function. When the following declaration is used outside any function:

```
bit init_flag;
```

init_flag will be globally visible.

Bit variables cannot be auto or parameters. A function may return a bit.

Bit variables behave in most respects like normal unsigned char variables, but they may only contain the values 0 and 1, and therefore provide a convenient and efficient method of storing boolean flags without consuming large amounts of internal RAM. It is not possible to declared pointers to *bit* variables or statically initialise *bit* variables. Operations on bits are performed using the single bit instructions wherever possible, thus the generated code to access bits is very efficient.

The entire bit psect is cleared on startup, but is not initialised. To create a bit object which has a non-zero initial value, explicitly initialise it at the beginning of your code.

Note that when assigning a larger integer type to a bit, only the least-significant bit is used. If you want to set a bit to be 0 or 1 depending on whether the other value is 0 or non-zero, use the form

```
bitvar = other_var != 0;
```

The bit psects are declared using the *bit* psect directive flag. Eight bit objects will take up one byte of storage space which is indicated by the bit psects' *scale* value of 8 in the map file. The length given in the map file for bit psects is in bits.

If the PICL flag `-STRICT` is used, the *bit* keyword becomes unavailable.

5.11.2.1 Using Bit-Addressable Registers

The *bit* variable facility may be combined with absolute variable declarations (see page 115) to access bits at specific addresses. Absolute bits are numbered from 0 (the least significant bit of the first byte) up. Therefore, bit number 3 in byte number 5 is actually absolute bit number 43 (that is 8bits/byte * 5 bytes + 3 bits).

For example, to access the Power Down Bit in the Status register, declare the status register to be at absolute address 03h, then declare a *bit* variable at absolute bit address 27:

```
static unsigned char STATUS @ 0x03;  
  
static bit PD @ (unsigned) &STATUS*8+3;
```

Note that all standard registers and bits are defined in the header files provided. The only header file you need to include to have access to the PIC registers is `<pic.h>` - at compile time this will include the appropriate header for the selected chip.

5.11.3 8-Bit Integer Data Types

HI-TECH C supports both *signed char* and *unsigned char* 8-bit integral types. The default *char* type is *unsigned char* unless the PICL `-SIGNED_CHAR` option is used, in which case it is *signed char*. *Signed char* is an 8-bit two's complement signed integer type, representing integral values from -128 to +127 inclusive. *Unsigned char* is an 8-bit unsigned integer type, representing integral values from 0 to 255 inclusive. It is a common misconception that the C *char* types are intended purely for ASCII character manipulation. This is not true, indeed the C language makes no guarantee that the default character representation is even ASCII. The *char* types are simply the smallest of up to four possible integer sizes, and behave in all respects like integers.

The reason for the name *char* is historical and does not mean that *char* can only be used to represent characters. It is possible to freely mix *char* values with *short*, *int* and *long* in C expressions. On the PIC the *char* types will commonly be used for a number of purposes, as 8-bit integers, as storage for ASCII characters, and for access to I/O locations. The default *unsigned char* type is the most efficient data type on the PIC and maps directly onto the 8-bit bytes which are most efficiently manipulated by PIC instructions. It is suggested that *char* types be used wherever possible so as to maximize performance and minimize code size.

5.11.4 16-Bit Integer Data Types

HI-TECH C supports four 16-bit integer types. *Int* and *short* are 16-bit two's complement signed integer types, representing integral values from -32,768 to +32,767 inclusive. *Unsigned int* and *unsigned short* are 16-bit unsigned integer types, representing integral values from 0 to 65,535 inclusive. All 16-bit integer values are represented in *little endian* format with the least significant byte at the lower address. Both *int* and *short* are 16 bits wide as this is the smallest integer size allowed by the ANSI standard for C. The sizes of the integer types were chosen so as not to violate the ANSI standard. Allowing a smaller integer size, such as 8 bits would lead to a serious incompatibility with the C standard. 8-bit integers are already fully supported by the *char* types and should be used in place of *int* wherever possible.

5.11.5 32-Bit Integer Data Types

HI-TECH C supports two 32-bit integer types. *Long* is a 32-bit two's complement signed integer type, representing integral values from -2,147,483,648 to +2,147,483,647 inclusive. *Unsigned long* is a 32-bit unsigned integer type, representing integral values from 0 to 4,294,967,295 inclusive. All 32-bit integer values are represented in *little endian* format with the least significant word and least significant byte at the lowest address. *Long* and *unsigned long* occupy 32 bits as this is the smallest long integer size allowed by the ANSI standard for C.

5.11.6 Floating Point

Floating point is implemented using the IEEE 754 32-bit format and a modified IEEE 754 (truncated) 24-bit format.

The truncated 24-bit format is used for all *float* values. For *double* values, the truncated 24-bit format is the default, but may be explicitly invoked with the PICL -D24 option. The 32-bit format is used for doubles by using the PICL -D32 option.

Both of these formats are described in Table 5 - 6, where:

- ☐ *sign* is the sign bit
- ☐ *exponent* is an 8-bit exponent which is stored as *excess 127* (i.e. an exponent of 0 is stored as 127)
- ☐ *mantissa* is the mantissa, which is to the right of the radix point. There is an implied bit to the left of the radix point which is always 1 except for a zero value, where the implied bit is zero. A zero value is indicated by a zero exponent.

The value of this number is $(-1)^{\text{sign}} \times 2^{(\text{exponent}-127)} \times 1.\text{mantissa}$.

Table 5 - 6 Floating Point Formats

Format	Sign	biased exponent	mantissa
IEEE 754 32-bit	x	xxxx xxxx	xxx xxxx xxxx xxxx xxxx xxxx
Modified IEEE 754 24-bit	x	xxxx xxxx	xxx xxxx xxxx xxxx

Here are some examples of the IEEE 754 32-bit and modified IEEE 754 24-bit formats:

Table 5 - 7 IEEE 754 32-bit and 24-bit Examples

Format	Number	biased exponent	1.mantissa	decimal
IEEE 754 32-bit	7DA6B69Bh	11111011b (251)	1.01001101011011010011011b (1.302447676659)	2.77000e+37
Modified IEEE 754 24-bit	42123Ah	10000100b (132)	1.001001000111010b (1.142395019531)	36.557

Note that the most significant bit of the mantissa column in Table 5 - 7 on page 114 (that is the bit to the left of the radix point) is the implied bit, which is assumed to be 1 unless the exponent is zero (in which case the float is zero).

The 32-bit example in Table 5 - 7 on page 114 can be calculated manually as follows.

The sign bit is zero; the biased exponent is 251, so the exponent is $251 - 127 = 124$. Take the binary number to the right of the decimal point in the mantissa. Convert this to decimal and divide it by 2^{23} where 23 is the number of bits taken up by the mantissa, to give 0.302447676659. Add one to this fraction. The floating-point number is then given by:

$$(-1)^0 \times 2^{(124)} \times 1.302447676659 = 1 \times 2.126764793256 \times 10^{37} \times 1.302447676659 \approx 2.77000 \times 10^{37}$$

5.12 Absolute Variables

A global or static variable can be located at an absolute address by following its declaration with the construct `@ address`, for example:

```
volatile unsigned char Portvar @ 0x06;
```

will declare a variable called *Portvar* located at 06h. Note that the compiler does not reserve any storage, but merely equates the variable to that address, the compiler generated assembler will include a line of the form:

```
_Portvar equ 06h
```

Note that the compiler and linker do not make any checks for overlap of absolute variables with other variables of any kind, so it is entirely the programmer's responsibility to ensure that absolute variables are allocated only in memory not in use for other purposes.

This construct is primarily intended for equating the address of a C identifier with a microprocessor register. To place a user-defined variable at an absolute address, define it in a separate psect and instruct the linker to place this psect at the required address. See "The #pragma psect Directive" on page 138.

5.13 Structures and Unions

HI-TECH C supports *struct* and *union* types of any size from one byte upwards. Structures and unions may be passed freely as function arguments and return values. Pointers to structures and unions are fully supported.

5.13.1 Structure Qualifiers

HI-TECH C supports the use of type qualifiers on structures. When a qualifier is applied to a structure, all of its members will attain this qualification. For example:

```
bank1 struct {
    int number;
    int *ptr;
} record;
```

In this example, the structure and its members *number* and *ptr* will be bank1 objects. Similarly, in the following example the structure is made constant.

```
const struct {
    int number;
    int *ptr;
} record;
```

In this case, the structure will be placed into ROM, however, if the members of the structure were individually made constant but the structure was not, then it would be positioned into RAM.

5.13.2 Bit Fields in Structures

HI-TECH C fully supports *bit fields* in structures.

Bit fields are allocated starting with the least significant bit of the word in which they will be stored. Bit fields are allocated within 8-bit words. The first bit allocated is the least significant bit of the byte. Bit fields are always allocated in 8-bit units. When a bit field is declared, it is allocated within the current 8-bit unit if it will fit, otherwise a new 8-bit byte is allocated within the structure. Bit fields never cross the boundary between 8-bit units allocation unit. For example, the declaration:

```
struct {
    unsigned      hi : 1;
    unsigned      dummy : 6;
    unsigned      lo : 1;
} foo @ 0x10;
```

will produce a structure occupying 1 byte from address 10h. The field *hi* will be bit 0 of address 10h, *lo* will be bit 7 of address 10h. The least significant bit of *dummy* will be bit 1 of address 10h and the most significant bit of *dummy* will be bit 6 of address 10h. If a bit field is declared in a structure that is assigned an absolute address, no storage will be allocated.

Unnamed bit fields may be declared to pad out unused space between active bits in control registers. For example, if *dummy* is never used the structure above could have been declared as:

```
struct {
    unsigned      hi : 1;
    unsigned      : 6;
    unsigned      lo : 1;
} foo @ 0x10;
```

5.14 Strings In ROM and RAM

An anonymous constant string is always placed in ROM and can only be accessed via a *const* pointer. In the following example, the string *Hello world* is a constant string and is stored in ROM. It is therefore accessed via a *const* pointer:

```
#define HELLO "Hello world"
SendBuff(HELLO);
```

A non-*const* array initialised with a string, for example:

```
char fred[] = "Hello world";
```

produces an array in RAM which is initialised at startup time with the string *Hello world* (copied from ROM), whereas a constant string used in other contexts represents an unnamed array qualified "const", accessed directly in ROM.

If you want to pass a constant string to a function argument, or assign it to a pointer, that pointer must be a *const char **, for example:

```
void SendBuff(const char * ptr)
```

or similar. Now you can pass either a pointer into ROM or RAM (on the midrange chips only - *const* pointers always point into ROM for baseline chips) and it will correctly fetch the data from the appropriate place.

5.15 Const and Volatile Type Qualifiers

HI-TECH C supports the use of the ANSI type qualifiers *const* and *volatile*.

The *const* type qualifier is used to tell the compiler that an object has a constant value and will not be modified. If any attempt is made to modify an object declared *const*, the compiler will issue a warning. User defined objects declared *const* are placed in a special psects in ROM. Obviously, a *const* object must be initialised when it is declared as it cannot be assigned a value at any point in the code following. For example:

```
const int version = 3;
```

The *volatile* type qualifier is used to tell the compiler that an object cannot be guaranteed to retain its value between successive accesses. This prevents the optimizer from eliminating apparently redundant references to objects declared *volatile* because it may alter the behaviour of the program to do so. All Input/Output ports and any variables which may be modified by interrupt routines should be declared *volatile*, for example:

```
volatile unsigned char P_A @ 0x05;
```

Volatile objects are accessed in a different way to non-volatile objects. For example, when assigning a non-volatile object the value 1, the object will be cleared and then incremented, but the same operation performed on a volatile object will load the W register with 1 and then store this to the appropriate address.

5.16 Placement and access of ROM objects

Objects stored in ROM include string literals, or constants, and any objects qualified using the *const* keyword. Placement of these objects varies on the device for which the code is compiled.

5.16.1 Midrange PICs

Midrange PICs store ROM-based objects as `retlw` instructions. Such objects are contained in psects called *strings* or *constn*, where n is a number. *Const* compound objects (for example structures or arrays) whose total size is less than 256 bytes, or *const* objects of basic type (for example `ints`) are stored in the *constn* psects. Other objects are stored in the *strings* psect. The *strings* psects is positioned explicitly whereas the *constn* psects are placed using the `CONST` class.

constn psects, in addition to the `retlw` instructions, begin with a `addwf pc` instruction. (This is why you cannot have exactly 256 bytes in *constn* psects.) This instruction is used by routines which perform indirect accessing of the objects.

5

5.17 Special Type Qualifiers

HI-TECH C supports special type qualifiers, *persistent*, *bank1*, *bank2* and *bank3* to allow the user to control placement of *static* and *extern* class variables into particular address spaces. If the PICL option, `-STRICT` is used, these type qualifiers are changed to `__persistent`, `__bank1`, `__bank2` and `__bank3`. These type qualifiers may also be applied to pointers. These type qualifiers may not be used on variables of class *auto*; if used on variables local to a function they must be combined with the *static* keyword. For example, you may not write:

```
void test(void)
{
    /* WRONG! */
    persistent int intvar;

    .. other code ..
}
```

because *intvar* is of class *auto*. To declare *intvar* as a *persistent* variable local to function *test()*, write:

```
static persistent int intvar;
```

5.17.1 Persistent Type Qualifier

By default, any C variables that are not explicitly initialised are cleared to zero on startup. This is consistent with the definition of the C language. However, there are occasions where it is desired for some data to be preserved across resets or even power cycles (on-off-on).

The *persistent* type qualifier is used to qualify variables that should not be cleared on startup. In addition, any *persistent* variables will be stored in a different area of memory to other variables (for example, the *nvr* or *nvr* psect).

5.17.2 Bank1, Bank2 and Bank3 Type Qualifiers

The *bank1*, *bank2* and *bank3* type qualifiers are used to place static variables in RAM Bank 1, RAM Bank 2 and RAM Bank 3 respectively. In the baseline microprocessors, pointers are unaffected by these type qualifiers.

Note that there is no *bank0* qualifier. Objects default to being in *bank0* if no other bank qualifier is used. All *auto* objects are positioned into Bank0, along with function parameters.

Here is an unsigned char in Bank 3:

```
static bank3 unsigned char fred;
```

Here is a pointer to an unsigned char in Bank 3:

```
bank3 unsigned char * ptrfred;
```

Here is another pointer to an unsigned char in Bank 3, except this time the pointer resides in Bank 2:

```
static bank 3 unsigned char * bank2 ptrfred;
```

5.18 Pointers

The format and use of pointers depend upon the range of processor.

5.18.1 Midrange Pointers

All pointers for the Midrange are the same as for the Baseline processors with the following exceptions:

- ☐ *RAM Pointers*
Because an 8-bit pointer can only access 256 bytes, RAM pointers can only access objects in Bank 0 and Bank 1.
- ☐ *Bank2 Pointers and Bank3 Pointers*
These pointers are RAM pointers which are used to access Bank 2 and Bank 3 of RAM respectively.

Note that at present it is not possible to have a Midrange RAM pointer which can access objects in three or more banks, or which can access objects in bank pairs other than those mentioned above.



Const Pointers

Const pointers for the Midrange processors are 16-bit wide. They can be used to access either ROM or RAM.

If the upper bit of the const pointer is non-zero, it is a pointer into RAM in any bank. A const pointer may be used to read from RAM locations, but writing to such locations is not permitted.

If the upper bit is zero, it is a pointer able to access the entire ROM space.

The ability of this pointer to access both ROM and RAM is very useful in string-related functions where a pointer passed to the function may point to a string in ROM or RAM.



Function Pointers

These pointers reference functions. A function is called using the address assigned to the pointer.

5.18.2 Combining Type Qualifiers and Pointers

5

The *const*, *volatile* and *persistent* modifiers may also be applied to pointers, controlling the behaviour of the object which the pointer addresses. When using these modifiers with pointer declarations, care must be taken to avoid confusion as to whether the modifier applies to the pointer, or the object addressed by the pointer. The rule is as follows: if the modifier is to the left of the “*” in the pointer declaration, it applies to the object which the pointer addresses. If the modifier is to the right of the “*”, it applies to the pointer variable itself. Using the *volatile* keyword to illustrate, the declaration:

```
volatile char * nptr;
```

declares a pointer to a *volatile* character. The *volatile* modifier applies to the object which the pointer addresses because it is to the left of the “*” in the pointer declaration.

The declaration:

```
char * volatile ptr;
```

behaves quite differently however. The *volatile* keyword is to the right of the “*” and thus applies to the actual pointer variable *ptr*, not the object which the pointer addresses. Finally, the declaration:

```
volatile char * volatile npnptr;
```

will generate a volatile pointer to a volatile variable.

5.18.3 Const Pointers

Pointers to *const* should be used when indirectly accessing objects which have been declared using the *const* qualifier. *Const* pointers behave in nearly the same manner as the default pointer class in each memory model, the only difference being that the compiler forbids attempts to write via a pointer to *const*. Thus, given the declaration:

```
const char *    cptr;
```

the statement:

```
ch = *cptr;
```

is legal, but the statement:

```
*cptr = ch;
```

is not. In the baseline series, *const* pointers always access program ROM because *const* declared objects are stored in ROM. In the midrange series, *const* pointers can access RAM as well as ROM.

5.19 Implementation-defined behaviour

Certain sections of the ANSI standard have implementation-defined behaviour. This section describes how the HI-TECH C compiler behaves in such situations.

5.19.1 Shifts applied to integral types

The ANSI standard states that the result of right shifting (>> operator) signed integral types is implementation defined. Typically, the possible actions that can be taken are that when an object is shifted right by one bit, the bit value shifted into the most significant bit of the result can either be zero, or a copy of the most significant bit before the shift took place. The latter case amounts to a sign extension of the number.

PICL performs a sign extension of any signed integral type (for example *signed char*, *signed int* or *signed long*). Thus an object with the signed int value 0124h shifted right one bit will yield the value 0012h and the value 8024h shifted right one bit will yield the value C012h.

Right shifts of unsigned integral values always clear the most significant bit of the result.

Left shifts (<< operator), signed or unsigned, always clear the least significant bit of the result.

5.19.2 Division and modulus with integral types

The sign of the result of division with integers when either operand is negative is implementation specific. Table 5 - 8 shows the expected sign of the result of the division of operand 1 with operand 2 when compiled with HI-TECH C.

Table 5 - 8 Integral division

Operand 1	Operand 2	Quotient	Remainder
+	+	+	+
-	+	-	-
+	-	-	+
-	-	+	-

In the case where the second operand is zero (division by zero), the result will always be zero.

5.20 Interrupt Handling in C

The compiler incorporates features allowing the PIC interrupt to be handled without writing any assembler code. The function qualifier *interrupt* may be applied to one function to allow it to be called directly from the hardware interrupt. The compiler will process the *interrupt* function differently to any other functions, generating code to save and restore any registers used and exit using the RETFIE instruction instead of a RETLW or RETURN instructions at the end of the function.

(If the PICL option -STRICT is used, the *interrupt* keyword becomes *__interrupt*. Wherever this manual refers to the *interrupt* keyword, assume *__interrupt* if you are using -STRICT.)

An *interrupt* function must be declared as type *interrupt void* and may not have parameters. It may not be called directly from C code, but it may call other functions itself, subject to certain limitations.

5.20.1 Midrange Interrupt Functions

An example of an *interrupt* function for a midrange PIC processor is shown here.

```
long    tick_count;
void interrupt tc_int(void)
{
    ++tick_count;
}
```

As there is a maximum of one interrupt vector in the midrange PIC series. Only one interrupt function may be defined. The interrupt vector will automatically be set to point to this function.

5.20.2 Context Saving on Interrupts

The PIC processor only saves the PC on its stack whenever an interrupt occurs. Other registers and objects must be saved in software. The HI-TECH C compiler determines which registers and objects are used by an interrupt function and saves these appropriately.

If the interrupt routine calls other functions and these functions are defined before the interrupt code in the same module, then any registers used by these functions will be saved as well. If the called functions have not been seen by the compiler, a worst case scenario is assumed and all registers and objects will be saved.

HI-TECH C does not scan assembly code which is placed in-line within the interrupt function for register usage. Thus, if you include in-line assembly code into an interrupt function, you may have to add extra assembly code to save and restore any registers or locations used.

5.20.2.1 MidRange Context Saving

The code associated with interrupt functions that do not require registers or objects is placed directly at the interrupt vector in a psect called *intcode*.

If context saving is required, this is performed by code placed in to a psect called *intentry* which will be placed at the interrupt vector. Any registers or objects to be saved are done so to areas of memory especially reserved for this purpose.

If the W register is to be saved, it is stored to memory reserved in the *intsave_0* psect which is located in Bank 0. If the processor for which the code is written has more than one RAM bank, it is impossible to swap to Bank 0 without corrupting W, so an *intsave_n* psect is allocated to each RAM bank (where *n* represents the bank number). The addresses of these memory areas are identical in the lower seven bits. When the interrupt occurs, W will be saved into one of these memory areas depending on the bank in which the processor was in before the interrupt occurred.

If the status register is to be saved, it is stored into memory reserved in the *intsave* psect which resides in Bank 0.

Some C code, for example division, may call an assembly routine which used temporary RAM locations. If these are used during the interrupt function, they too will be saved by separate routines which are automatically linked.

5.20.3 Context Retrieval

Any objects saved by the compiler are automatically restored before the interrupt function returns. Midrange PIC restoration code is placed into a psect called *int_ret*.

5.20.4 Interrupt Levels

Normally it is assumed by the compiler that any interrupt may occur at any time, and an error will be issued by the linker if a function appears to be called by an interrupt and by main-line code, or another interrupt. Since it is often possible for the user to guarantee this will not happen for a specific routine, the compiler supports an interrupt level feature.

This is achieved with the `#pragma interrupt_level` directive. There are two interrupt levels available, and any interrupt routines at the same level will be assumed by the compiler to be mutually exclusive. (Since the midrange PIC devices only support one active interrupt there is no value in using more than one interrupt level when these processors are selected.) This exclusion must be guaranteed by the user - the compiler is not able to control interrupt priorities. Each interrupt routine may be assigned a single level, either 0 or 1.

In addition, any non-interrupt routines that are called from an interrupt function and also from main-line code may also use the `#pragma interrupt_level` directive to specify that they will never be called by interrupts of one or more levels. This will prevent linker from issuing an error message because the function was included in more than one call graph. Note that it is entirely up to the user to ensure that the function is NOT called by both main-line and interrupt code at the same time. This will normally be ensured by disabling interrupts before calling the function. It is not sufficient to disable interrupts inside the function after it has been called.

An example of using the interrupt levels is given below. Note that the `#pragma` directive applies to only the immediately following function. Multiple `#pragma interrupt_level` directives may precede a non-interrupt function to specify that it will be protected from multiple interrupt levels.

```
5  /* a non-interrupt function called at interrupt time and by main-
   line code */
   #pragma interrupt_level 1
   void bill(){
       int i;
       i = 23;
   }

   /* two interrupt functions calling the same non-interrupt
   function */
   #pragma interrupt_level 1

   void interrupt fred(void)
   {
       bill();
   }

   #pragma interrupt_level 1
   void interrupt joh()
   {
       bill();
   }
```

```

    }

    main()
    {
        bill();
    }

```

5.21 Mixing C and Assembler Code

Assembly language code can be mixed with C code using three different techniques.

5.21.1 External Assembly Language Functions

Entire functions may be coded in assembly language as separate *.as* source files, assembled by the assembler (ASPIC) and combined into the binary image using the linker. This technique allows arguments and return values to be passed between C and assembler code. To access an external function, first include an appropriate C *extern* declaration in the calling C code. For example, suppose you need an assembly language function to provide access to the rotate left through carry instruction on the PIC:

```
extern char    rotate_left(char);
```

declares an external function called *rotate_left()* which has a return value type of *char* and takes a single argument of type *char*. The actual code for *rotate_left()* will be supplied by an external *.as* file which will be separately assembled with ASPIC. The full PIC assembler code for *rotate_left()* would be something like:

```

processor      16C84

psect    text0,class=CODE,local,delta=2
global   _rotate_left
signat   _rotate_left,4201
_rotate_left
; Fred is passed in the W register - assign it
; to ?a_rotate_left.
movwf    ?a_rotate_left

; Rotate left. The result is placed in the W register.
rlf      ?a_rotate_left,w

; The return is already in the W register as required.
return

```

```
FNSIZE    _rotate_left,1,0
global    ?a_rotate_left
end
```

The name of the assembly language function is the name declared in C, with an underscore prepended. The *global* pseudo-op is the assembler equivalent to the C *extern* keyword and the *signat* pseudo-op is used to enforce link time calling convention checking. Signature checking and the *signat* pseudo-op are discussed in more detail later in this chapter.

Note that in order for assembly language functions to work properly they must look in the right place for any arguments passed and must correctly set up any return values. Local variable allocation (via the *fsize* directive), argument and return value passing mechanisms are discussed in detail later in the manual and should be understood before attempting to write assembly language routines.

5.21.2 Accessing C objects from within assembler

Global C objects may be directly accessed from within assembly code using their name prepended with an underscore character. For example, the object `foo` defined globally in a C module:

```
int foo;
```

may be access from assembler as follows.

```
global    _foo
movwf     _foo
```

If the assembler is contained in a different module, then the `global` assembler directive should be used in the assembler code to make the symbol name available, as above. If the object is being accessed from in-line assembly in another module, then an *extern* declaration for the object can be made in the C code, for example:

```
extern int foo;
```

This declaration will only take effect in the module if the object is also accessed from within C code. If this is not the case then, an in-line global assembler directive should be used. Care should be taken in the object is defined in a bank other than 0. The address of a C object includes the bank information which must be stripped before the address can be used in PIC instructions. Failure to do this may result in fixup errors issued by the linker. For example, if the object `fred` is defined for the PIC16C77 processor as follows:

```
bank2 int fred;
```

then writing the value 0x55 to `fred` via assembler could be performed as follows.

```

movlw    055h
bcf      3,5
bsf      3,6 ; select bank 2
movwf    _fred&07Fh

```

The PIC16C77 processor has banks which are 0x80 bytes long and hence has a 7-bit field in its instructions for RAM addresses. The mask value used, in this case 0x7F, may be different if other processors are used. Check your PIC programmer's guide for information relating to the bank sizes and instruction formats.

If in doubt as to writing assembler which access C objects, write code in C which performs a similar task to what you intend to do and study the assembler listing file produced by the compiler.

5.21.3 #asm, #endasm and asm()

PIC instructions may also be directly embedded in C code using the directives *#asm*, *#endasm* and *asm()*. The *#asm* and *#endasm* directives are used to start and end a block of assembler instructions which are to be embedded inside C code. The *asm()* directive is used to embed a single assembler instruction in the code generated by the C compiler. To continue our example from above, you could directly code a rotate left on a memory byte using either technique as the following example shows:

```

#include <stdio.h>
unsigned char var;
void main(void)
{
    var = 1;

    #asm
        rlf _var,f
    #endasm
    asm("rlf _var,f");
}

```

When using inline assembler code, great care must be taken to avoid interacting with compiler generated code. If in doubt, compile your program with the PICL -S option and examine the assembler code generated by the compiler.

IMPORTANT NOTE: the *#asm* and *#endasm* construct is not syntactically part of the C program, and thus it does *not* obey normal C flow-of-control rules. For example, you cannot use a *#asm* block with an if statement and expect it to work correctly. If you use in-line assembler around any C constructs such as if, while, do etc. they you should use only the *asm("")* form, which is a C statement and will correctly interact with all C flow-of-control structures.

5.22 Signature Checking

The compiler automatically produces signatures for all functions. A signature is a 16-bit value computed from a combination of the function's return data type, the number of its parameters and other information affecting the calling sequence for the function. This signature is output in the object code of any function referencing or defining the function.

At link time the linker will report any mismatch of signatures. Thus if a function is declared in one module in a different way (for example, as returning a *char* instead of *short*) then the linker will report an error.

It is sometimes necessary to write assembly language routines which are called from C using an *extern* declaration. Such assembly language functions need to include a signature which is compatible with the C prototype used to call them. The simplest method of determining the correct signature for a function is to write a dummy C function with the same prototype and compile it to assembly language using the PICL -S option. For example, suppose you have an assembly language routine called `_widget` which takes two *int* arguments and returns a *char* value. The prototype used to call this function from C would be:

```
extern char widget(int, int);
```

Where a call to `_widget` is made in the C code, the signature for a function with two *int* arguments and a *char* return value would be generated. In order to match the correct signature the source code for `widget` needs to contain an ASPIC *SIGNAT* pseudo-op which defines the same signature value. To determine the correct value, you would write the following code:

```
char widget(int arg1, int arg2)
{
}
```

and compile it to assembler code using

```
PICL -S x.c
```

The resultant assembler code includes the following line:

```
signat _widget,8297
```

The *SIGNAT* pseudo-op tells the assembler to include a record in the *.OBJ* file which associates the value 8297 with symbol `_widget`. The value 8297 is the correct signature for a function with two *int* arguments and a *char* return value. If this line is copied into the *.AS* file where `_widget` is defined, it will associate the correct signature with the function and the linker will be able to check for correct argument passing. For example, if another *.C* file contains the declaration:

```
extern char widget(long);
```

then a different signature will be generated and the linker will report a signature mis-match which will alert you to the possible existence of incompatible calling conventions.

5.23 Linking Programs

The compiler will automatically invoke the linker unless requested to stop after producing assembler code (PICL -S option) or object code (PICL -C option).

PICL and HTLPIC by default generate *Intel HEX* files and *Bytecraft COD*. If you use the -BIN option or specify an output file with a .BIN filetype using the PICL -O option the compiler will generate a binary image instead. After linking, the compiler will automatically generate a memory usage map which shows the address used by, and the total sizes of, all the memory areas which are used by the compiled code. Note that bit objects are shown separately. For example:

Memory Usage Map:

Program ROM	\$0000 - \$001A	\$001B (27) words	
Program ROM	\$07EE - \$07FF	\$0012 (18) words	
		\$002D (45) words total	Program ROM
Bank 0 RAM	\$0020 - \$0022	\$0003 (3) bytes total	Bank 0 RAM
Bank 1 RAM	\$00A0 - \$00A2	\$0003 (3) bytes total	Bank 1 RAM
Bank 0 Bits	\$0118 - \$0119	\$0002 (2) bits total	Bank 0 Bits

More detailed memory usage information, listed in ascending order of individual psects, may be obtained by using the PICL -PSECTMAP option.

5.24 Memory Usage

The compiler makes few assumptions about memory. With the exception of variables declared using the @address construct, absolute addresses are not allocated until link time.

The memory used is based upon page and bank information in the chipinfo file (which defaults to lib\picinfo.ini) The linker will automatically locate code and ROM data into all the available memory pages and ensure that a psect does not straddle a page boundary.

5.25 Register Usage

In the midrange processors, the W register is used for register-based function argument passing and for function return values. This register should be preserved by any assembly language routines which are called.

5.26 Function Argument Passing

The method used to pass function arguments depends on the size of the argument or arguments.

If there is only one argument, and it is one byte in size, it is passed in the W register.

If there is only one argument, and it is greater than one byte in size, it is passed in the argument area of the called function. If there are subsequent arguments, these arguments are also passed in the argument area of the called function.

If there is more than one argument, and the first argument is one byte in size, it is passed in the auto variable area of the called function, with subsequent arguments being passed in the argument area of the called function.

In the case of a variable argument list, which is defined by the ellipsis symbol (...), the calling function builds up the variable argument list and passes a pointer to the variable part of the argument list in *btemp*. *btemp* is the label at the start of the *temp* psect (i.e. the psect used for temporary data).

Take, for example, the following ANSI-style function:

```
void test(int a, int b, int c)
{
}
```

The function *test()* will receive all arguments in its function argument block. A call:

```
test( 0x65af, 0x7288, 0x080c);
```

would generate code similar to:

```
movlw    0AFh
movwf    ((?_test))&7fh)
movlw    065h
movwf    ((?_test+1))&7fh)
movlw    088h
movwf    ((0+((?_test)+02h))&7fh)
movlw    072h
movwf    ((1+((?_test)+02h))&7fh)
movlw    0Ch
movwf    ((0+((?_test)+04h))&7fh)
movlw    08h
movwf    ((1+((?_test)+04h))&7fh)
lcall    (_test)
```


Parameters passed to a function are referred to by a label which consists of question mark (?) followed by an underscore (_) and the name of the function to which is added an offset. So, for example in the above code, the first parameter to the function `test`, the int value `0x65af`, is held in the memory locations `?_test` and `?_test+1`.

It is often helpful to write a dummy C function with the same argument types as your assembler function, and compile to assembler code with the PICL -S option, allowing you to examine the entry and exit code generated. In the same manner, it is useful to examine the code generated by a call to a function with the same argument list as your assembler function.

5.27 Function Return Values

Function return values are passed to the calling function as follows:

5.27.1 8-Bit Return Values

For the baseline processors, 8-bit values (*char*, *unsigned char* and *pointer*) are returned in memory via the *temp* psect.

For the midrange processors, 8-bit values are returned in the W register. For example, the function:

```
char return_8(void)
{
    return 0;
}
```

will exit with the following code:

```
movlw    0
return
```

5.27.2 16-Bit and 32-bit Return Values

16-bit and 32-bit values (*int*, *unsigned int*, *short*, *unsigned short* and some *pointer* values; *long*, *unsigned long*, *float* and *double*) are returned in memory, with the least significant word in the lowest memory location. For example, the function:

```
int return_16(void)
{
    return 0x1234;
}
```

will exit with the following code:

```
    movlw    low 01234h
    movwf    btemp
    movlw    high 01234h
    movwf    btemp+1
    return
```

5.27.3 Structure Return Values

Composite return values (*struct* and *union*) of size 4 bytes or smaller are returned in memory as with 16-bit and 32-bit return values. For composite return values of greater than 4 bytes in size, the structure or union is copied into the *struct* psect. For example:

```
struct fred
{
    int ace[4];
}

struct fred return_struct(void)
{
    struct fred wow;

    return wow;
}
```

will exit with the following code:

```
    movlw    ?a_return_struct+0
    movwf    4
    movlw    structret
    movwf    btemp
    movlw    8
    global   structcopy
    lcall    structcopy
    return
```

5.28 Function Calling Convention

The baseline PIC devices have a two-level deep hardware stack which is used to store the return address of a subroutine call. Typically, PICL uses a call instruction to transfer control to a C function when it is called, however on baseline processors, the size of the stack severely limits the level of nested C function calls possible.

By default, function calls on baseline PICs are implemented using a method involving an assembly jump instruction and a lookup table, or jump table. A function is “called” by jumping directly to its address after storing the address of a jump table instruction which will be able to return control back to the calling function. The address is stored as an object local to the function being called. The lookup table is accessed after the function called has finished executing. This method allows functions to be nested without overflowing the stack, however it does come at the expense of memory and program speed.

To disable the lookup-table mode of operation, a function definition can be qualified as *fastcall*, so that calls to this function are performed using the usual call assembly instruction. Extreme care must be used when functions are declared as *fastcall*, since the each nested *fastcall* will use one word of available stack space. Check the call graph in the map file to ensure that the stack will not overflow.

The function prototype for a baseline *fastcall* function might look something like:

```
fastcall void my_function(int a);
```

The midrange and high end PIC devices have larger stacks and are thus allow a higher degree of function nesting. These devices do not use the lookup table method when calling functions.

5.29 Local Variables

C supports two classes of local variables in functions: *auto* variables which are normally allocated in the function’s auto variable block and *static* variables which are always given a fixed memory location.

5.29.1 Auto Variables

Auto (i.e. automatic) variables are the default type of local variable. Unless explicitly declared to be *static* a local variable will be made *auto*. Auto variables are allocated in the auto variable block and referenced by indexing off the start of that function’s block. The variables will not necessarily be allocated in the order declared - in contrast to parameters which are always in lexical order. Note that most type qualifiers cannot be used with auto variables, since there is no control over the storage location. Exceptions are *const* and *volatile*.

All auto variables are allocated memory in bank 0. The bank qualifiers cannot be used with objects of type *auto*.

The auto variable blocks for a number of functions are overlapped by the linker if those functions are never called at the same time.

Auto objects are referenced with a symbol that consists of a question mark (?) concatenated with *a_function* plus some offset, where *function* is the name of the function in which the object is defined. For example, if the *int* object *test* is the first local object defined in the function *main* it will be accessed using the addresses *?a_main* and *?a_main+1*.

5.29.2 Static Variables

Uninitialized *static* variables are allocated in the *rbss_n* psect and occupy fixed memory locations which will not be overlapped by storage for other functions. Static variables are local in scope to the function which they are declared in, but may be accessed by other functions via pointers. Static variables are guaranteed to retain their value between calls to a function, unless explicitly modified via a pointer. Static variables are not subject to any architectural limitations on the PIC.

Static variables which are initialised are only done so once during the program's execution. Thus, they may be preferable over initialised auto objects which are assigned a value every time the block in which the definition is placed is executed.

5.30 Compiler Generated Psects

The compiler splits code and data objects into a number of standard program sections (referred to as *psects*). The HI-TECH assembler allows an arbitrary number of named psects to be included in assembler code. The linker will group all data for a particular psect into a single segment.

If you are using PICL or HTLPIC to invoke the linker, you don't need to worry about the information documented here, except as background knowledge. If you want to run the linker manually (this is not recommended), or write your own assembly language subroutines, you should read this section carefully.

❑ The compiler generated psects which are placed in ROM are:

- powerup* Which contains executable code for the standard or user-supplied power-up routine.
- idata_n* These psects (where *n* is the bank number) contain the ROM image of any initialised variables. These psects are copied into the *rdata_n* psects at startup.
- textn* These psects (where *n* is a number) contain all executable code for the Midrange and High-end processors. They also contains any executable code after the first *goto* instruction which can never be skipped for the Baseline processors.
- ctextn* These psects (where *n* is a number) are used only in the Baseline processors. They contain executable code from the entry point of each *fastcall* function until the first *goto* instruction which can never be skipped. Further executable code is placed in the *textn* psects.
- text* Is a global psect used for executable code for some library functions.
- constn* These psects (where *n* is a number) hold objects that are declared *const* and which are not modifiable.
- strings* The *strings* psect is used for some *const* objects. Const objects whose size exceeds 256 bytes, for example const arrays, are positioned in this psect. It also includes all unnamed string

constants, such as string constants passed as arguments to routines like *printf()* and *puts()*. This psect is linked into ROM, since it does not need to be modifiable.

stringtable

The *stringtable* psect contains the string table which is used to access objects in the *strings* psect. This psect will only be generated if there is a *strings* or baseline *jmp_tab* psect.

jmp_tab Only for the Baseline processors, this is another strings psect used to store jump addresses and function return values.

config Used to store the config word.

idloc Used to store the ID location words.

intentry Contains the entry code for the interrupt service routine. This code saves the necessary registers and parts of the *temp* psect.

intcode Is the psect which contains the executable code for the interrupt service routine.

intret Is the psect which contains the executable code responsible for restoring saved registers and objects after an interrupt routine has completed executing.

init Used by initialisation code which, for example, clears RAM.

end_init Used by initialisation code which, for example, clears RAM.

float_text Used by some library routines, and in particular by arithmetic routines. It is possible that this psect will have a non-zero length even if no floating point operations are included in a program.

clrtext Used by some startup routines for clearing the *rbss_n* psests.

□ The compiler generated psests which are placed in RAM are:

rbss_n These psests (where *n* is the bank number) contain any uninitialized variables.

rdata_n These psests (where *n* is the bank number) contain any initialised variables.

nvrn This psect is used to store *persistent* variables. It is not cleared or otherwise modified at startup.

rbit_n These psests (where *n* is the bank number) are used to store all *bit* variables except those declared at absolute locations. The declaration:

```
static bit flag;
```

will allocate *flag* as a single bit in the *rbit* psect.

- struct* Contains any structure which is returned from a function.
- intsave* Holds the *W* register saved by the interrupt service routine. If necessary, the *W* register will also be saved in the *intsave_n* psects.
- intsave_n* (Where *n* is a non-zero bank number) may also hold the *W* register saved by the interrupt service routine. (See the description of the *intsave* psect.)
- temp* Is used to store scratch variables used by the compiler. These include function return values larger than a *char* and values passed to and returned from library routines. If possible, this will be positioned in the common area of the processor.
- xtemp* Is used to store scratch variables used by the compiler and to pass values to and from the library routines.

5.31 Runtime Startoff Modules

The starting address of a C program is usually the lowest code address (0h). The global symbol *powerup* is at this address. The baseline PIC devices, the code at the start address may perform some initialisation, notably clearing of the *bss* (uninitialized data) psect and copying initialised data (which is not declared *const*) into RAM.

The startup code calls the function *_main*. Note the underscore; prepended to the function name - all symbols derived from external names in a C program have this character prepended by the compiler which helps prevent conflict with assembler names. The function *main()* is, by definition of the C language, the “main program”.

The run-time startup code is provided by a standard module found in the *lib* directory, chosen from one of those listed in Table 5 - 9.

Table 5 - 9 Standard Run-time Startoff Modules

Startoff Module	Processor supported
<i>picrt400.obj</i>	Midrange processor, 1 ROM bank, 2 RAM banks

The source code used to generate all of the runtime startoff modules is called *picrt66x.as* which is in the *sources* directory. In addition to this will be the routines to copy data or clear memory as required. The sources for these additional routines are contained in files, for example *clrbank0.as* (to clear bank 0 memory) and *cpybank1.as* (to copy the bank 1 data psect).

5.31.1 The *powerup* Routine

Some hardware configurations require special initialisation, often within the first few cycles of execution after reset. Rather than having to modify the run-time startoff module to achieve this there is

a hook to the reset vector provided via the *powerup* routine. This is a user-supplied assembler module that will be executed immediately on reset. Often this can be embedded in a C module as embedded assembler code. A “dummy” *powerup* routine is included in the file *powerup.as*. The file can be copied, modified and included into your project. It will replace the default *powerup* routine.

The *powerup* routine should be written assuming that little or no RAM is working and should only use system resources after it has tested and enabled them. The following example code shows the default *powerup* routine which are in the standard library:

```
global powerup, start
psect powerup, class=CODE, delta=2

powerup

    ljmp    start
```

The *powerup* routine is generally intended to be relatively small, however, since it is linked before the interrupt vectors, it may interfere with them if it becomes too large. To avoid conflicting with the interrupt vectors, the *powerup* routine can be made to jump to a separate function, which will be linked at a different location, and then jumps to *start*. The following gives an example of this:

```
global powerup, start, big_powerup
psect powerup, class=CODE, delta=2

powerup

    ljmp    big_powerup

psect big_powerup, class=CODE, delta=2

big_powerup

    ...powerup code...

    ljmp start
```

5.32 Linker-Defined Symbols

The link address of a *psect* can be obtained from the value of a global symbol with name `__Lname` where *name* is the name of the *psect*. For example, `__Lbss` is the low bound of the *bss* *psect*. The highest address of a *psect* (i.e. the link address plus the size) is symbol `__Hname`. If the *psect* has different load and link addresses, as may be the case if the *data* *psect* is linked for RAM operation, the load address is `__Bname`.

5.33 Preprocessor Directives

The HI-TECH C compiler accepts several specialised preprocessor directives in addition to the standard directives. These are listed in Table 5 - 10 on page 139.

Macro expansion using arguments can use the `#` character to convert an argument to a string, and the `##` sequence to concatenate tokens.

5.34 Pragma Directives

There are certain compile-time directives that can be used to modify the behaviour of the compiler. These are implemented through the use of the ANSI standard *pragma* facility. The format of a pragma is:

```
#pragma keyword options
```

where *keyword* is one of a set of keywords, some of which are followed by certain options. A list of the keywords is given in Table 5 - 11 on page 140. Those keywords not discussed elsewhere are detailed below.

5.34.1 The `#pragma jis` and `nojis` Directives

If your code includes strings with two-byte characters in the JIS encoding for Japanese and other national characters, the `#pragma jis` directive will enable proper handling of these characters, specifically not interpreting a back-slash (`\`) character when it appears as the second half of a two byte character. The `nojis` directive disables this special handling. JIS character handling is disabled by default.

5.34.2 The `#pragma printf_check` Directive

Certain library functions accept a format string followed by a variable number of arguments in the manner of `printf()`. Although the format string is interpreted at run-time, it can be compile-time checked for consistency with the remaining arguments. This directive enables this checking for the named function, e.g. the system header file `<stdio.h>` includes the directive `#pragma printf_check(printf) const` to enable this checking for `printf()`. You may also use this for any user-defined function that accepts `printf`-style format strings. The qualifier following the function name is to allow automatic conversion of pointers in variable argument lists. The above example would cast any pointers to strings in RAM to be pointers of the type `(const char *)`.

Note that the warning level must be set to -1 or below for this option to have effect.

5.34.3 The `#pragma psect` Directive

Normally the object code generated by the compiler is broken into the standard psects as already documented. This is fine for most applications, but sometimes it is necessary to redirect variables or code into different psects when a special memory configuration is desired. Code and data for any of the

Table 5 - 10 Preprocessor directives

Directive	Meaning	Example
#	preprocessor null directive, do nothing	#
#assert	generate error if condition false	#assert SIZE > 10
#asm	generate error if condition false	#asm movlw 10h #endasm
#define	define preprocessor macro	#define SIZE 5 #define FLAG #define add(a,b) ((a)+(b))
#elif	short for #else #if	see #ifdef
#else	conditionally include source lines	see #if
#endasm	terminate in-line assembly	see #asm
#endif	terminate conditional source inclusion	see #if
#error	generate an error message	#error Size too big
#if	include source lines if constant expression true	#if SIZE < 10 c = process(10) #else skip(); #endif
#ifdef	include source lines if preprocessor symbol defined	#ifdef FLAG do_loop(); #elif SIZE == 5 skip_loop(); #endif
#ifndef	include source lines if preprocessor symbol not defined	#ifndef FLAG jump(); #endif
#include	include text file into source	#include <stdio.h> #include "project.h"
#line	specify line number and file name for listing	#line 3 final
#nn	(where <i>nn</i> is a number) short for #line <i>nn</i>	#20
#pragma	compiler specific options	See section 5.34 on page 138
#undef	undefines preprocessor symbol	#undef FLAG

standard C psects may be redirected using a `#pragma psect` directive. For example, if all the uninitialised global data in a particular C source file is to be placed into a psect called *otherram*, the following directive should be used:

```
#pragma psect rbss_0=otherram
```

This directive tells the compiler that anything which would normally be placed in the *rbss_0* psect should now be placed in the *otherram* psect.

Table 5 - 11 Pragma Directives

Directive	Meaning	Example
interrupt_level	Allow interrupt function to be called from main-line code. See section 5.20.4 on page 123	#pragma interrupt_level 2
jis	Enable JIS character handling in strings	#pragma jis
nojis	Disable JIS character handling (default)	#pragma nojis
printf_check	Enable printf-style format string checking	#pragma printf_check(printf) const
psect	Rename compiler-defined psect	#pragma psect text=mytext
regsused	Specify registers which are used in an interrupt	#pragma regsused w

Placing code in a different psect is slightly different. Code is placed in a multiple psects which have names like: *text0*, *text1*, *text2* and so on. Thus you do not know the exact psect in which code will reside. To redirect code the following preprocessor directive can be used.

```
#pragma psect text%%u=othercode
```

The `%u` sequence corresponds to the internal representation of the text psect number. The additional percent character is used to ensure that the psect name is scanned correctly.

Any given psect should only be redirected once in a particular source file, and all psect redirections for a particular source file should be placed at the top of the file, below any `#include` statements and above any other declarations. For example, to declare a group of uninitialized variables which are all placed in a psect called *otherram*, the following technique should be used:

```
--File OTHERRAM.C
#pragma psect rbss_0=otherram
char buffer[5];
int var1, var2, var3;
```

Any files which need to access the variables defined in *otherram.c* should #include the following header file:

```
--File OTHERRAM.H
extern char  buffer[5];
extern int   var1, var2, var3;
```

The *#pragma psect* directive allows code and data to be split into arbitrary memory areas. Definitions of code or data for non-standard psects should be kept in separate source files as documented above. When linking code which uses non-standard psect names, you will need to use the PICL -L option to specify an extra linker option, or use the linker manually, or use an HTLPIC project to compile and link your code. If you want a nearly standard configuration with the addition of only an extra psect like *otherram*, you can use the PICL -L option to add an extra -P specification to the linker command. For example:

```
PICL -L-Potherram=50h/400h -16C84 test.obj otherram.obj
```

will link *test.obj* and *otherram.obj* with a standard configuration, and the extra *otherram* psect at 50h in RAM, but not overlapping any valid ROM load address. If you are using the HTLPIC integrated environment you can set up a project file by selecting **Start New Project**, add the names of your four source files using **Source Files ...** and then modify the linker options to include any new psects by selecting **Linker Options ...**.

5.34.4 The #pragma regsused Directive

HI-TECH C will automatically save context when an interrupt occurs. The compiler will determine only those registers and objects which need to be saved for the particular interrupt function defined. The *#pragma regsused* directive allows the programmer to further limit the registers and objects that the compiler might save and retrieve on interrupt.

Table 5 - 12 on page 142 shows registers names that would commonly be used with this directive. The register names are not case sensitive and a warning will be produced if the register name is not recognised.

This pragma affects the first interrupt function following in the source code.

For example, to limit the compiler to saving no registers other than the W register and FSR register for an interrupt function, use:

```
#pragma regsused w fsr
```

Even if a register, other than W or FSR, has been used and that register would normally be saved, it will not be saved if this pragma is in effect. The W and/or FSR register will only be automatically saved by the compiler if required.

Table 5 - 12 Valid regsused Register Names

Register Name	Description
w	W register
btemp, btemp+1...btemp+11	btemp temporary area
fsr	indirect data pointer
tblreg	table registers: low and high byte of table pointer and table latch

5.35 Standard I/O Functions and Serial I/O

A number of the standard I/O functions are provided in the C library with the compiler, specifically those functions intended to read and write formatted text on standard output and input. A list of the available functions is in Table 5 - 13.

Table 5 - 13 Supported STDIO Functions

Function name	Purpose
printf(const char * s, ...)	Formatted printing to stdout
sprintf(char * buf, const char * s, ...)	Writes formatted text to buf

Before any characters can be written or read using these functions, the *putch()* and *getch()* functions must be written. Other routines which may be required include *getche()* and *kbhit()*.

You will find samples of serial code which implements the *putch()* and *getch()* functions in the *samples\serial* directory.

5.36 MPLAB-specific Debugging Information

Certain options and compiler features are specifically intended to help MPLAB perform symbolic debugging. The **-FAKELOCAL** switch (**Fake local symbols**) performs two functions, both specific to MPLAB. Since MPLAB does not read the local symbol information produced by the compiler, this options generates additional global symbols which can be used to represent local symbols in a program. The format for the symbols is *function_name.symbol_name*. Thus, if a variable called *foo* was defined inside the function *main*, MPLAB would allow access to a global object called *main.foo*. This symbol format is not available in assembler code. References to this object in assembler would be via the symbol *_main\$foo*. Although this information allows access to most local objects, if there are two or more objects with the same name in the same function, then you will not be able to examine both as they redefine the same symbol.

The **-FAKELOCAL** switch also alters the line numbering information produced so that MPLAB can better follow the C source when performing source-level stepping.

The PICL Macro Assembler

The HI-TECH PICL Macro Assembler assembles source files for the selected Microchip PIC microprocessors (16C84, 16F84, 16F84A & 16F627). This chapter describes the usage of the assembler and the directives (assembler pseudo-ops and controls) accepted by the assembler.

The PICL Macro Assembler package includes a linker, librarian, cross reference generator and an object code converter.

6.1 Assembler Usage

The assembler is called ASPIC and is available to run on PC and Unix machines. Note that the assembler will not produce any messages unless there are errors or warnings - there are no “assembly completed” messages.

The usage of the assembler is similar under all of these operating systems. All command line options are recognised in either upper or lower case. The basic command format is shown:

```
aspic [ options ] files ...
```

Files is a space separated list of one or more assembler source files. Where more than one source file is specified the assembler treats them as a single module, i.e. a single assembly will be performed on the concatenation of all the source files specified. The files must be specified in full, no default extensions or suffixes are assumed.

Options is an optional space separated list of assembler options, each with a minus sign (-) as the first character. A full list of possible options is given in Table 6 - 1 on page 154, and a full description of each option follows.

6.2 Assembler Options

The command line options recognised by ASPIC are as follows:

-processor

This option defines the processor which is being used. Table 4 - 3 on page 94 shows the options which are available. Note that this table will be added to as new processors become available. You can also add your own processors to the compiler. For more information about this, See “Processor Support” on page 105..

-A

An assembler file with an extension *.opt* will be produced if this option is used. This is useful when checking the optimized assembler produced using the *-O* option.

Table 6 - 1 ASPIC Assembler options

Option	Meaning	Default
<i>-processor</i>	Define the processor	
<i>-A</i>	Produce assembler output	Produce object code
<i>-C</i>	Produce cross-reference	No cross reference
<i>-Cchipinfo</i>	Define the chipinfo file	lib\lite84.ini
<i>-Eformat</i>	Set error format	
<i>-Flength</i>	Specify listing form length	66
<i>-H</i>	Output hex values for constants	Decimal values
<i>-I</i>	List macro expansions	Don't list macros
<i>-Llistfile</i>	Produce listing	No listing
<i>-O</i>	Perform optimization	No optimization
<i>-Ooutfile</i>	Specify object name	srcfile.OBJ
<i>-Raddress</i>	Maximum ROM size	
<i>-S</i>	No size error messages	
<i>-U</i>	No undefined symbol messages	
<i>-V</i>	Produce line number info	No line numbers
<i>-Wwidth</i>	Specify listing page width	80
<i>-X</i>	No local symbols in OBJ file	

-C A cross reference file will be produced when this option is used. This file, called *srcfile.crf* where *srcfile* is the base portion of the first source file name, will contain raw cross reference information. The cross reference utility CREF must then be run to produce the formatted cross reference listing.

-Cchipinfo Define the chipinfo file to use. This option is not normally required as the chipinfo file *lib\lite84.ini* is normally not used.

-E The default format for an error message is in the form:

```
filename: line: message
```

where the error of type *message* occurred on line *line* of the file *filename*. The *-E2* option will produce a less-readable format which is used by HPD.

-Flength The default listing pagelength is 66 lines (11 inches at 6 lines per inch). The *-F* option allows a different page length to be specified.

-
- H Particularly useful in conjunction with the -A option, this option specifies to output constants as hexadecimal values rather than decimal values.
 - I This option forces listing of macro expansions and unassembled conditionals which would otherwise be suppressed by a NOLIST assembler control. The -L option is still necessary to produce a listing.
 - Llistfile This option requests the generation of an assembly listing. If *listfile* is specified then the listing will be written to that file, otherwise it will be written to the standard output.
 - O This requests the assembler to perform optimisation on the assembly code. Note that the use of this option slows the assembly down, as the assembler must make an additional pass over the input code.
 - Ooutfile By default the assembler determines the name of the object file to be created by stripping any suffix or extension (i.e. the portion after the last dot) from the first source file name and appending *.obj*. The -O option allows the user to override the default and specify an explicit filename for the object file.
 - Raddress The value, *address*, passed to the assembler with this option is the highest address used by code in ROM. From this value, the assembler can determine how many page bits need to be adjusted for an `fcall` or `ljmp` pseudo instruction.
 - S If a byte-size memory location is initialized with a value which is too large to fit in 8 bits, then the assembler will generate a “Size error” message. Use of the -S option will suppress this type of message.
 - U Undefined symbols encountered during assembly are treated as external, however an error message is issued for each undefined symbol unless the -U option is given. Use of this option suppresses the error messages only, it does not change the generated code.
 - V This option will include in the object file produced by the assembler line number and file name information for the use of a debugger. Note that the line numbers will be assembler code lines - when assembling a file produced by the compiler, there will be *line* and *file* directives inserted by the compiler so this option is not required.
 - Wwidth This option allows specification of the listfile paper width, in characters. *Width* should be a decimal number greater than 41. The default width is 80 characters.
 - X The object file created by the assembler contains symbol information, including local symbols, i.e. symbols that are neither public or external. The -X option will prevent the local symbols from being included in the object file, thereby reducing the file size.

6.3 PICL Assembly Language

The source language accepted by the HI-TECH Software PICL Macro Assembler is described below. All opcode mnemonics and operand syntax are strictly PIC assembly language. Additional mnemonics are documented in this section.

6.3.1 Additional Mnemonics

Apart from the PIC assembly language mnemonics, the PICL assembler includes the *fcall* and *ljmp* mnemonics. These instructions implement *call* and *goto* instructions but with the added job of setting the necessary bits in PCLATH. These additional mnemonics should be used where possible as they make assembler code independent of the final position of the routines that are to be executed.

6.3.2 Assembler Format Deviations

The HI-TECH PICL assembler uses a slightly modified form of assembly language to that specified by Microchip. Certain PIC instructions used by Microchip assembler use the operands “,0” or “,1” to specify the destination for the result of that operation. The HI-TECH PICL assembler uses the more-readable operands “,w” and “,f” to specify the destination register. The W register is selected as the destination when using the “,w” operand, and the file register is selected when using the “,f” operand or if no destination operand is specified. The case of the letter in the destination operand is not important. The Microchip numerical operands cannot be used with the HI-TECH PICL assembler.

6.3.3 Character Set

The character set used is standard 7 bit ASCII. Alphabetic case is significant for identifiers, but not opcodes and reserved words. Tabs are treated as equivalent to spaces.

6.3.4 Constants

6.3.4.1 Numeric Constants

The assembler performs all arithmetic as signed 32 bit. Errors will be caused if a quantity is too large to fit in a memory location. The default radix for all numbers is 10. Other radices may be specified by a trailing base specifier as given in Table 6 - 2.

Table 6 - 2 ASPIC Numbers and bases

Radix	Format
Binary	digits 0 and 1 followed by <i>B</i>
Octal	digits 0 to 7 followed by <i>O</i> , <i>Q</i> , <i>o</i> or <i>q</i>
Decimal	digits 0 to 9 followed by <i>D</i> , <i>d</i> or nothing
Hexadecimal	digits 0 to 9, A to F preceded by <i>Ox</i> or followed by <i>H</i> or <i>h</i>

Hexadecimal numbers must have a leading digit (e.g. 0ffffh) to differentiate them from identifiers. Hexadecimal constants are accepted in either upper or lower case.

Note that a binary constant must have an upper case B following it, as a lower case b is used for temporary (numeric) label backward references.

In expressions, real numbers are accepted in the usual format, and are interpreted as IEEE 32-bit format. A real number may be converted into the truncated IEEE 24-bit format by using the float24 pseudo-function. Here is an example of its use:

```
movlw    low(float24(31.415926590000002))
```

6.3.4.2 Character Constants

A character constant is a single character enclosed in single quotes ('). Multi character constants may be specified using double quotes. See “Strings” on page 160.

6.3.5 Delimiters

All numbers and identifiers must be delimited by white space, non-alphanumeric characters or the end of a line.

6.3.6 Special Characters

There are a few characters that are special in certain contexts. Within a macro body, the character @ is used for token concatenation. In a macro argument list, the angle brackets < and > are used to quote macro arguments.

6.3.7 Identifiers

Identifiers are user-defined symbols representing memory locations or numbers. A symbol may contain any number of characters drawn from the alphabets, numerics and the special characters dollar (\$), question mark (?) and underscore(_). The first character of an identifier may not be numeric. The case of alphabets is significant, e.g. *Fred* is not the same symbol as *fred*. Some examples of identifiers are shown here:

```
An_identifier
an_identifier
an_identifier1
$$$
?$_12345
```

6.3.7.1 Significance of Identifiers

Users of other assemblers that attempt to implement forms of data typing for identifiers should note that this assembler attaches no significance to any symbol, and places no restrictions or expectations on the

usage of a symbol. The names of *psects* (program sections) and ordinary symbols occupy separate, overlapping name spaces, but other than this, the assembler does not care whether a symbol is used to represent bytes, words or chicken sheds. No special syntax is needed or provided to define the addresses of bits or any other data type, nor will the assembler issue any warnings if a symbol is used in more than one context. The instruction and addressing mode syntax provide all the information necessary for the assembler to generate correct code.

6.3.7.2 Assembler-Generated Identifiers

Where a LOCAL directive is used in a macro block, the assembler will generate a unique symbol to replace each specified identifier in each expansion of that macro. These unique symbols will have the form `??nnnn` where *nnnn* is a 4 digit number. The user should avoid defining symbols with the same form.

6.3.7.3 Location Counter

The current location within the active program section is accessible via the symbol `$`.

6.3.7.4 Register Symbols

The working register (W) is available by using its standard name, and case of the register name is not significant. Special Function Registers (SFRs) are available using their standard names, as long as they are already defined. The standard SFRs for each selected PIC processor are defined in the corresponding header file.

It is not possible to equate a symbol to a register.

6.3.7.5 Labels

A label is a name at the beginning of a statement which is assigned a value equal to the current offset within the current psect (program section). There are two types of labels; symbol labels and numeric labels.

A label is not the same as a macro name, which also appears at the beginning of the line in a macro declaration.

6.3.7.6 Symbolic Labels

A symbolic label may be any symbol, and may or may not be followed by a colon. Here are two examples of legitimate labels:

```
frank
simon44:
```

Symbols not interpreted in any other way are assumed to be labels. Thus the code:

```
movlw 23h
```

```

    bananas
    movf 37h

```

defines a symbol called `bananas`. Mis-typed assembler instructions can sometimes be treated as labels without an error message being issued. Indentation of a label does not force the assembler to treat it as an mnemonic.

6.3.7.7 Numeric Labels

The assembler implements a system of numeric labels (as distinct from the local labels used in macros) which relieves the programmer from creating new labels within a block of code. A numeric label is a numeric string followed by a colon, and may be referenced by the same numeric string with either an ‘f’ or ‘b’ suffix.

When used with an ‘f’ suffix, the label reference is the first label with the same number found by looking forward from the current location, and conversely a ‘b’ will cause the assembler to look *backward* for the label.

For example:

```

    _entry_point                                ; Referenced from somewhere
    else
    1
        .
        .
        .
        decfsz _counter
        goto 1b
        goto 1f
        .
        .
        .
    1                                           ; End of the function
        return
    end

```

Note that even though there are two 1: labels, no ambiguity occurs, since each is referred to uniquely. The *goto 1b* refers to a label further back in the source code, while *goto 1f* refers to a label further forward. In general, to avoid confusion, it is recommended that within a routine you do not duplicate numeric labels.

6.3.8 Strings

A string is a sequence of characters not including carriage return or newline, enclosed within matching quotes. Either single (') or double (") quotes may be used, but the opening and closing quotes must be the same. A string used as an operand to a DB directive may be any length, but a string used as operand to an instruction must not exceed 1 or 2 characters, depending on the size of the operand required.

6.3.9 Expressions

Expressions are made up of numbers, symbols, strings and operators. Operators can be unary (one operand, e.g. *not*) or binary (two operands, e.g. *+*). The operators allowable in expressions are listed in Table 6 - 3. The usual rules governing the syntax of expressions apply.

The operators listed may all be freely combined in both constant and relocatable expressions. The HI-TECH linker permits relocation of complex expressions, so the results of expressions involving relocatable identifiers may not be resolved until link time.

6.3.10 Statement Format

Legal statement formats are shown in Table 6 - 4. The second form is only legal with certain directives, such as MACRO, SET and EQU. The *label* field is optional and if present should contain one identifier. The *name* field is mandatory and should also contain one identifier. Note that a label, if present, may or may not be followed by a colon. There is no limitation on what column or part of the line any part of the statement should appear in.

6.3.11 Program Sections

Program sections, or *psects*, are a way of grouping together parts of a program even though the source code may not be physically adjacent in the source file, or even where spread over several source files. Unless defined as ABS (absolute), psects are relocatable.

A psect is identified by a name and has several attributes. The psect directive is used to define psects. It takes as arguments a name and an optional comma-separated list of flags. See the section PSECT on page 164 for full information. The assembler associates no significance to the name of a psect.

The following is an example showing some executable instructions being placed in the *text0* psect, and some data being placed in the *rbss_0* psect.

```
processor 16C84

psect  text0,class=CODE,local,delta=2
adjust
lcall  clear_fred
movf   flag
btfss  3,2
```

Table 6 - 3 Operators

Operator	Purpose
*	Multiplication
+	Addition
-	Subtraction
/	Division
= or eq	Equality
> or gt	Signed greater than
>= or ge	Signed greater than or equal to
< or lt	Signed less than
<= or le	Signed less than or equal to
<> or ne	Signed not equal to
low	Low byte of operand
high	High byte of operand
highword	High 16 bits of operand
mod	Modulus
&	Bitwise AND
^	Bitwise XOR (exclusive or)
	Bitwise OR
not	Bitwise complement
<< or shl	Shift left
>> or shr	Shift right
rol	Rotate left
ror	Rotate right
seg	Segment (bank number) of address
float24	24-bit version of real operand
nul	Tests if macro argument is null

Table 6 - 4 ASPIC Statement formats

Format 1:	label	opcode	operands	; comment
Format 2:	name	pseudo-op	operands	; comment
Format 3:	; comment only			

goto 1f
incf fred

```

1      goto    2f
      decf    fred
2

      psect   rbss_0,class=BANK0,space=1
flag
      ds      1
fred
      ds      1

      psect   text0,class=CODE,local,delta=2
clear_fred
      clrf    fred
      bcf     status,5
      return

```

Note that even though the two blocks of code in the *text0* psect are separated by a block in the *rbss_0* psect, the two *text0* psect blocks will be contiguous when loaded by the linker. In other words, the `decf fred` instruction will fall through to the label `clear_fred` during execution. The actual location in memory of the two psects will be determined by the linker. See the linker manual for information on how psect addresses are determined.

A label defined in a psect is said to be relocatable, that is, its actual memory address is not determined at assembly time. Note that this does not apply if the label is in the default (unnamed) psect, or in a psect declared absolute (see the PSECT pseudo-op description below). Any labels declared in an absolute psect will be absolute, that is their address will be determined by the assembler.

Relocatable expressions may be combined freely in expressions.

6.3.12 Assembler Directives

Assembler directives, or *pseudo-ops*, are used in a similar way to opcodes, but either do not generate code, or generate non-executable code, i.e. data bytes. The directives are listed in Table 6 - 5 on page 163, and are detailed below.

6.3.12.1 GLOBAL

GLOBAL declares a list of symbols which, if defined within the current module, are made public. If the symbols are not defined in the current module, it is a reference to symbols in external modules. Example:

```
GLOBAL lab1,lab2,lab3
```


Table 6 - 5 ASPIC Directives (pseudo-ops)

Directive	Purpose
GLOBAL	Make symbols accessible to other modules or allow reference to other modules'
END	End assembly
PSECT	Declare or resume program section
ORG	Set location counter
EQU	Define symbol value
DEFL	Define or re-define symbol value
DB	Define constant byte(s)
DW	Define constant word(s)
DS	Reserve storage
IF	Conditional assembly
ELSIF	Alternate conditional assembly
ELSE	Alternate conditional assembly
ENDIF	End conditional assembly
FNADDR	Inform linker that a function may be indirectly called
FNARG	Inform linker that evaluation of arguments for one function requires calling another
FNBREAK	Break call graph links
FNCALL	Inform linker that one function calls another
FNCONF	Supply call graph configuration info for linker
FNINDIR	Inform linker that all functions with a particular signature may be indirectly called
FNROOT	Inform linker that a function is the "root" of a call graph
FNSIZE	Inform linker of argument and local variable sizes for a function
MACRO	Macro definition
ENDM	End macro definition
LOCAL	Define local tabs
ALIGN	Align output to the specified boundary
PAGESEL	Generate set/reset instruction to set PCLATH for this page
PROCESSOR	Define the particular chip for which this file is to be assembl.ed.
REPT	Repeat a block of code n times
IRP	Repeat a block of code with a list
IRPC	Repeat a block of code with a character list
SIGNAT	Define function signature

6.3.12.2 END

END is optional, but if present should be at the very end of the program. It will terminate the assembly. If an expression is supplied as an argument, that expression will be used to define the start address of the program. Whether this is of any use will depend on the linker. Example:

```
END start_label
```

6.3.12.3 PSECT

The PSECT directive declares or resumes a program section. It takes as arguments a name and optionally a comma separated list of flags. The allowed flags are listed in Table 6 - 6 below. Once a psect has been declared it may be resumed later by simply giving its name as an argument to another psect directive; the flags need not be repeated.

Table 6 - 6 PSECT flags

Flag	Meaning
ABS	Psect is absolute
BIT	Psect holds bit objects
CLASS	Specify class name for psect
DELTA	Size of an addressing unit
GLOBAL	Psect is global (default)
LIMIT	Upper address limit of psect
LOCAL	Psect is not global
OVRLD	Psect will overlap same psect in other modules
PURE	Psect is to be read-only
RELOC	Start psect on specified boundary
SIZE	Maximum size of psect
SPACE	Represents area in which psect will reside
WITH	Place psect in the same page as specified psect

- ☐ ABS defines the current psect as being absolute, i.e. it is to start at location 0. This does not mean that this module’s contribution to the psect will start at 0, since other modules may contribute to the same psect.
- ☐ The BIT flag specifies that a psect hold objects that are 1 bit long. Such psects have a *scale* value of 8 to indicate that there are 8 addressable units to each byte of storage.
- ☐ The CLASS flag specifies a class name for this psect. Class names are used to allow local psects to be referred to by a class name at link time, since they cannot be referred to by their own name. Class names are also useful where the linker address range feature is to be used.

- ☐ The DELTA flag defines the size of an addressing unit. In other words, the number of bytes covered for an increment in the address. This should be DELTA=2 for ROM (i.e. a word) and DELTA=1 (which is the default delta value) for RAM.
- ☐ A psect defined as GLOBAL will be combined with other global psects of the same name from other modules at link time. GLOBAL is the default.
- ☐ The LIMIT flag specifies a limit on the highest address to which a psect may extend.
- ☐ A psect defined as LOCAL will not be combined with other local psects at link time, even if there are others with the same name. A local psect may not have the same name as any global psect, even one in another module.
- ☐ A psect defined as OVRLD will have the contribution from each module overlaid, rather than concatenated at run time. OVRLD in combination with ABS defines a truly absolute psect, i.e. a psect within which any symbols defined are absolute.
- ☐ The PURE flag instructs the linker that this psect will not be modified at run time and may therefore, for example, be placed in ROM. This flag is of limited usefulness since it depends on the linker and target system enforcing it.
- ☐ The RELOC flag allows specification of a requirement for alignment of the psect on a particular boundary, e.g. RELOC=100h would specify that this psect must start on an address that is a multiple of 100h.
- ☐ The SIZE flag allows a maximum size to be specified for the psect, e.g. SIZE=100h. This will be checked by the linker after psects have been combined from all modules.
- ☐ The SPACE flag is used to differentiate areas of memory which have overlapping addresses, but which are distinct. Psects which are positioned in ROM and RAM have a different SPACE value to indicate that ROM address zero, for example, is a different location to RAM address zero. Objects in different RAM banks have the same SPACE value as their full addresses (including bank information) are unique.
- ☐ The WITH flag allows a psect to be placed in the same page *with* a specified psect. For example WITH=text0 will specify that this psect should be placed in the same page as the *text0* psect.

Some examples of the use of the PSECT directive follow:

```
PSECT    fred
PSECT    bill, size=100h, global
PSECT    joh, abs, ovrlld, class=CODE, delta=2
```

6.3.12.4 ORG

ORG changes the value of the location counter within the current psect. This means that the addresses set with ORG are relative to the base of the psect, which is not determined until link time.

The argument to ORG must be either an absolute value, or a value referencing the current psect. In either case the current location counter is set to the value of the argument. For example:

```
ORG      100h
```

will move the location counter to the beginning of the current psect + 100h. The actual location will not be known until link time. It is possible to move the location counter backwards.

In order to use the ORG directive to set the location counter to an absolute value, an absolute, overlaid psect must be used:

```
psect    absdata, abs, ovrl, delta=2
org      addr
```

where *addr* is an absolute address.

6.3.12.5 EQU

This pseudo-op defines a symbol and equates its value to an expression. For example

```
thomas EQU 123h
```

The identifier *thomas* will be given the value 123h. EQU is legal only when the symbol has not previously been defined. See also SET on page 166

6.3.12.6 SET

This pseudo-op is equivalent to EQU except that allows a symbol to be re-defined. For example

```
thomas SET 0h
```

6.3.12.7 DEFL

DEFL (define label) is identical to EQU except that it may be used to re-define a symbol.

6.3.12.8 DB

DB is used to initialize storage as bytes. The argument is a list of expressions, each of which will be assembled into one byte. Each character of the string will be assembled into one memory location.

An error will occur if the value of an expression is too big to fit into the memory location, e.g. if the value 1020 is given as an argument to DB.

Examples:

```
alabel DB 'X',1,2,3,4,
```

Note that because the size of an address unit in ROM is 2 bytes, the DB pseudo-op will initialise a word with the upper byte set to zero.

6.3.12.9 DW

DW operates in a similar fashion to DB, except that it assembles expressions into words. An error will occur if the value of an expression is too big to fit into a word.

Example:

```
DW -1, 3664h, 'A', 3777Q
```

6.3.12.10 DS

This directive reserves, but does not initialize, memory locations. The single argument is the number of bytes to be reserved. Examples:

```
alabel: DS      23          ;Reserve 23 bytes of memory
xlabel: DS      2+3        ;Reserve 5 bytes of memory
```

6.3.12.11 FNADDR

This directive tells the linker that a function has its address taken, and thus could be called indirectly through a function pointer. For example

```
FNADDR _func1
```

tells the linker that *func1()* has its address taken.

6.3.12.12 FNARG

The directive

```
FNARG fun1, fun2
```

tells the linker that evaluation of the arguments to function *fun1* involves a call to *fun2*, thus the memory argument memory allocated for the two functions should not overlap. For example, the C function calls

```
fred(var1, bill(), 2);
```

will generate the assembler directive

```
FNARG _fred, _bill
```

thereby telling the linker that *bill()* is called while evaluating the arguments for a call to *fred()*.

6.3.12.13 FNBREAK

This directive is used to break links in the call graph information. The form of this directive is as follows:

```
FNBREAK fun1, fun2
```

and is automatically generated when the `interrupt_level` pragma is used. It states that the link to `fun1` in the call graph rooted at `fun2` should not be followed when checking for functions that appear in multiple call graphs. `Fun2` is typically `intlevel0` or `intlevel1` in compiler generated code.

6.3.12.14 FNCALL

This directive takes the form:

```
FNCALL fun1, fun2
```

FNCALL is usually used in compiler generated code. It tells the linker that function *fun1* calls function *fun2*. This information is used by the linker when performing call graph analysis. If you write assembler code which calls a C function, use the FNCALL directive to ensure that your assembler function is taken into account. For example, if you have an assembler routine called *_fred* which calls a C routine called *foo()*, in your assembler code you should write:

```
FNCALL _fred, _foo
```

6.3.12.15 FNCONF

The FNCONF directive is used to supply the linker with configuration information for a *call graph*. FNCONF is written as follows:

```
FNCONF psect, auto, args
```

where `psect` is the psect containing the call graph, `auto` is the prefix on all *auto* variable symbol names and `args` is the prefix on all function argument symbol names. This directive normally appears in only one place: the runtime startoff code used by C compiler generated code. For the HI-TECH PIC Compiler the PICRT66X.AS routine should include the directive:

```
FNCONF rbss, ?a, ?
```

telling the linker that the call graph is in the *rbss* psect, auto variable blocks start with *?a* and function argument blocks start with *?*.

6.3.12.16 FNINDIR

This directive tells the linker that a function performs an indirect call to another function with a particular signature (see the SIGNAT directive). The linker must assume worst case that the function could call any other function which has the same signature and has had its address taken (see the FNADDR directive). For example, if a function called *fred()* performs an indirect call to a function with signature 8249, the compiler will produce the directive:

```
FNINDIR _fred, 8249
```

6.3.12.17 FNSIZE

The FNSIZE directive informs the linker of the size of the local variable and argument area associated with a function. These values are used by the linker when building the call graph and assigning addresses to the variable and argument areas. This directive takes the form:

```
FNSIZE func, local, args
```

The named function has a local variable area and argument area as specified, for example

```
FNSIZE _fred, 10, 5
```

means the function *fred()* has 10 bytes of local variables and 5 bytes of arguments. The function name arguments to any of the call graph associated directives may be local or global. Local functions are of course defined in the current module, but must be used in the call graph construction in the same manner as global names.

6.3.12.18 FNROOT

This directive tells the assembler that a function is a “root function” and thus forms the root of a call graph. It could either be the C *main()* function or an interrupt function. For example, the C main module produce the directive:

```
FNROOT _main
```

6.3.12.19 IF, ELSIF, ELSE and ENDIF

These directives implement conditional assembly. The argument to IF and ELSIF should be an absolute expression. If it is non-zero, then the code following it up to the next matching ELSE will be assembled. If the expression is zero then the code up to the next matching ELSE will be skipped.

At an ELSE the sense of the conditional compilation will be inverted, while an ENDIF will terminate the conditional assembly block. Example:

```
IF      ABC
lcall   aardvark
ELSIF   DEF
lcall   denver
ELSE
lcall   grapes
ENDC
```

In this example, if *ABC* is non-zero, the first *lcall* instruction will be assembled but not the second or third. If *ABC* is zero and *DEF* is non-zero, the second *lcall* will be assembled but the first and third will

not. If both *ABS* and *DEF* are zero, the third *lcall* will be assembled. Conditional assembly blocks may be nested.

6.3.12.20 MACRO and ENDM

These directives provide for the definition of macros. The **MACRO** directive should be preceded by the macro name and optionally followed by a comma separated list of formal parameters. When the macro is used, the macro name should be used in the same manner as a machine opcode, followed by a list of arguments to be substituted for the formal parameters.

For example:

```
;macro: swap
;args:  arg1, arg2 - the NUMBERS of the variables to swap
;      arg3 - the NAME of the variable to use for temp storage;
;descr: Swaps two specified variables, where the variables
;      are named:
;              var_x
;      and x is a number.
;uses:  Uses the w register.

swap    macro    arg1, arg2, arg3
        movf     var_@arg1,w
        movwf    arg3
        movf     var_@arg2,w
        movwf    var_@arg1
        movf     arg3,w
        movwf    var_@arg2
        endm
```

When used, this macro will expand to the 3 instructions in the body of the macro, with the formal parameters substituted by the arguments. Thus:

```
swap    2,4,tempvar
```

expands to:

```
movf     var_2,w
movwf    tempvar
movf     var_4,w
movwf    var_2
movf     tempvar,w
movwf    var_4
```


A point to note in the above example: the @ character is used to permit the concatenation of macro parameters with other text, but is removed in the actual expansion. The NUL operator may be used within a macro to test a macro argument, for example:

```

    if      nul      arg3                ; argument was not supplied.
        ...
    else
        ...                            ; argument was supplied
    endif

```

A comment may be suppressed within the expansion of a macro (thus saving space in the macro storage) by opening the comment with a double semicolon (; ;).

6.3.12.21 LOCAL

The LOCAL directive allows unique labels to be defined for each expansion of a given macro. Any symbols listed after the LOCAL directive will have a unique assembler generated symbol substituted for them when the macro is expanded. For example:

```

down    macro    count
        local    more
        movlw    count
        movwf    tempvar
more     decfsz   tempvar
        goto     more
        endm

```

when expanded will include a unique assembler generated label in place of *more*. For example:

```

down    4

```

expands to:

```

        movlw    4
        movwf    tempvar
??0001 decfsz    tempvar
        goto     ??0001

```

if invoked a second time, the label *more* would expand to *??0002*.

6.3.12.22 ALIGN

The `ALIGN` directive aligns whatever is following, data storage or code etc., to the specified boundary in the psect in which the directive is found. The boundary is specified by a number following the directive and is a number of bytes. For example, to align output to a 2 byte (even) address within a psect, the following could be used.

```
ALIGN 2
```

Note, however, that what follows will only begin on an even absolute address if the psect begins on an even address. The `ALIGN` directive can also be used to ensure that a psect's length is a multiple of a certain number. For example, if the above `ALIGN` directive was placed at the end of a psect, the psect would have a length that was always an even number of bytes long.

6.3.12.23 REPT

The `REPT` directive temporarily defines an unnamed macro then expands it a number of times as determined by its argument. For example:

```
rept    3
addwf   fred, fred
andwf   fred, w
endm
```

will expand to

```
addwf   fred, fred
andwf   fred, w
addwf   fred, fred
andwf   fred, w
addwf   fred, fred
andwf   fred, w
```

6.3.12.24 IRP and IRPC

The `IRP` and `IRPC` directives operate similarly to `REPT`. However, instead of repeating the block a fixed number of times, it is repeated once for each member of an argument list. In the case of `IRP` the list is a conventional macro argument list, in the case of `IRPC` it is each character in one argument. For each repetition the argument is substituted for one formal parameter.

For example:

```
psect   idata_0

irp      number, 4865h, 6C6Ch, 6F00h
```

```
dw    number
endm
```

```
psect text0
```

would expand to:

```
psect idata_0
```

```
dw    4865h
```

```
dw    6C6Ch
```

```
dw    6F00h
```

```
psect text0
```

Note that you can use *local* labels and angle brackets in the same manner as with conventional macros.

The IRPC directive is similar, except it substitutes one character at a time from a string of non-space characters.

For example:

```
psect idata_0
```

```
irpc  char,ABC
```

```
dw    char
```

```
endm
```

```
psect text0
```

will expand to:

```
psect idata_0
```

```
dw    'A'
```

```
dw    'B'
```

```
dw    'C'
```

```
psect text0
```

6.3.12.25 PAGESEL

It's sometimes necessary to set the current PCLATH bits so that a goto will jump to a location in the current page of ROM. The LJMP and FCALL instructions automatically generate the necessary code to set or reset the PCLATH bits, but at other times an explicit directive PAGESEL is used, e.g.

```
PAGESEL      $
```

6.3.12.26 PROCESSOR

The output of the assembler depends on which chip it is desired to assemble for. This can be set on the command line, or with this directive, e.g.

```
PROCESSOR    16C84
```

6.3.12.27 SIGNAT

This directive is used to associate a 16-bit signature value with a label. At link time the linker checks that all signatures defined for a particular label are the same and produces an error if they are not. The SIGNAT directive is used by the HI-TECH C compiler to enforce link time checking of function prototypes and calling conventions.

Use the SIGNAT directive if you want to write assembly language routines which are called from C. For example:

```
SIGNAT  _fred, 8192
```

will associate the signature value 8192 with symbol `_fred`. If a different signature value for `_fred` is present in any object file, the linker will report an error.

6.3.13 Macro Invocations

When invoking a macro, the argument list must be comma separated. If it is desired to include a comma (or other delimiter such as a space) in an argument then angle brackets (< and >) may be used to quote the argument. In addition the exclamation mark (!) may be used to quote a single character. The character immediately following the exclamation mark will be passed into the macro argument even if it is normally a comment indicator.

If an argument is preceded by a percent sign (%), that argument will be evaluated as an expression and passed as a decimal number, rather than as a string. This is useful if evaluation of the argument inside the macro body would yield a different result.

6.3.14 Assembler Controls

Assembler controls may be included in the assembler source to control such things as listing format. These keywords have no significance anywhere else in the program. Some keywords are followed by one or more parameters.

A list of keywords is given in Table 6 - 7, and each is described further below.

Table 6 - 7 ASPIC Assembler controls

Control ^a	Meaning	Format
COND*	Include conditional code in the listing	cond
EXPAND	Expand macros in the listing output	expand
INCLUDE	Textually include another source file	include <pathname>
LIST*	Define options for listing output	list [<listopt>, ..., <listopt>]
NOCOND	Leave conditional code out of the listing	nocond
NOEXPAND*	Disable macro expansion	noexpand
NOLIST	Disable listing output	nolist
NOXREF*	Disable the cross-reference listing	noxref
PAGE	Start a new page in the listing output	page
SPACE	Insert blank lines into the listing output	space <no.lines>
SUBTITLE	Specify the subtitle of the program	subtitle "<subtitle>"
TITLE	Specify the title of the program	title "<title>"
XREF	Generate a cross-reference listing	xref

a.The default options are listed with an asterisk (*).

6.3.14.1 COND

Any conditional code will be included in the listing output. See also the NOCOND control.

6.3.14.2 EXPAND

When EXPAND is in effect, the code generated by macro expansions will appear in the listing output. See also the NOEXPAND control.

6.3.14.3 INCLUDE

This control causes the file specified by `pathname` to be textually included at that point in the assembly file. The INCLUDE control must be the last control keyword on the line.

6.3.14.4 LIST

If the listing was previously turned off using the NOLIST control, the LIST control on its own will turn the listing on.

Alternatively, the LIST control may includes options to control the assembly and the listing. The options are listed in Table 6 - 8. See also the NOLIST control.

Table 6 - 8 LIST Control Options

List Option	Default	Description
c=nnn	80	Set the page (i.e. column) width.
n=nnn	59	Set the page length.
t=ON OFF	OFF	Truncate listing output lines. The default wraps lines.
p=<processor>	n/a	Set the processor type. The options are listed in Table 4 - 3 on page 94.
r=<radix>	hex	Set the default radix to hex, dec or oct.
x=ON OFF	OFF	Turn macro expansion on or off.

6.3.14.5 NOCOND

Any conditional code will not be included in the listing output. See also the COND control.

6.3.14.6 NOEXPAND

NOEXPAND disables macro expansion in the listing file. The macro call will be listed instead. See also the EXPAND control.

6.3.14.7 NOLIST

This control turns the listing output off from this point onwards. See also the LIST control.

6.3.14.8 NOXREF

NOXREF will disable generation of the *raw* cross reference file. See also the XREF control.

6.3.14.9 PAGE

PAGE causes a new page to be started in the listing output. A *Control-L* (form feed) character will also cause a new page when encountered in the source.

6.3.14.10 SPACE

The SPACE control will place the a number of blank lines in the listing output as specified by its parameter.

6.3.14.11 SUBTITLE

SUBTITLE defines a subtitle to appear at the top of every listing page, but under the title. The string should be enclosed in single or double quotes. See also the TITLE control.

6.3.14.12 TITLE

This control keyword defines a title to appear at the top of every listing page. The string should be enclosed in single or double quotes. See also the SUBTITLE control.

6.3.14.13 XREF

XREF is equivalent to the command line option `-CR`; it causes the assembler to produce a *raw* cross reference file. The utility CREF should be used to actually generate the *formatted* cross-reference listing.

Linker and Utilities Reference Manual

7.1 Introduction

HI-TECH C incorporates a relocating assembler and linker to permit separate compilation of C source files. This means that a program may be divided into several source files, each of which may be kept to a manageable size for ease of editing and compilation, then each source file may be compiled separately and finally all the object files linked together into a single executable program.

This chapter describes the theory behind and the usage of the linker. Note however that in most instances it will not be necessary to use the linker directly, as the compiler drivers (HPD or command line) will automatically invoke the linker with all necessary arguments. Using the linker directly is not simple, and should be attempted only by those with a sound knowledge of the compiler and linking in general.

If it is absolutely necessary to use the linker directly, the best way to start is to copy the linker arguments constructed by the compiler driver, and modify them as appropriate. This will ensure that the necessary startup module and arguments are present.

Note also that the linker supplied with HI-TECH C is generic to a wide variety of compilers for several different processors. Not all features described in this chapter are applicable to all compilers.

7.2 Relocation and Pssects

The fundamental task of the linker is to combine several relocatable object files into one. The object files are said to be *relocatable* since the files have sufficient information in them so that any references to program or data addresses (e.g. the address of a function) within the file may be adjusted according to where the file is ultimately located in memory after the linkage process. Thus the file is said to be relocatable. Relocation may take two basic forms; relocation by name, i.e. relocation by the ultimate value of a global symbol, or relocation by pssect, i.e. relocation by the base address of a particular section of code, for example the section of code containing the actual executable instructions.

7.3 Program Sections

Any object file may contain bytes to be stored in memory in one or more program sections, which will be referred to as *pssects*. These pssects represent logical groupings of certain types of code bytes in the program. In general the compiler will produce code in three basic types of pssects, although there will be several different types of each. The three basic kinds are *text* pssects, containing executable code, *data* pssects, containing initialised data, and *bss* pssects, containing uninitialised but reserved data.

The difference between the data and bss psects may be illustrated by considering two external variables; one is initialised to the value 1, and the other is not initialised. The first will be placed into the data psect, and the second in the bss psect. The bss psect is always cleared to zeros on startup of the program, thus the second variable will be initialised at run time to zero. The first will however occupy space in the program file, and will maintain its initialised value of 1 at startup. It is quite possible to modify the value of a variable in the data psect during execution, however it is better practice not to do so, since this leads to more consistent use of variables, and allows for restartable and romable programs.

For more information on the particular psects used in a specific compiler, refer to the appropriate machine-specific chapter.

7.4 Local Psects

Most psects are *global*, i.e. they are referred to by the same name in all modules, and any reference in any module to a global psect will refer to the same psect as any other reference. Some psects are *local*, which means that they are local to only one module, and will be considered as separate from any other psect even of the same name in another module. Local psects can only be referred to at link time by a class name, which is a name associated with one or more psects via a *CLASS=* directive in assembler code.

7.5 Global Symbols

The linker handles only symbols which have been declared as global to the assembler. From the C source level, this means all names which have storage class external and which are not declared as static. These symbols may be referred to by modules other than the one in which they are defined. It is the linker's job to match up the definition of a global symbol with the references to it. Other symbols (local symbols) are passed through the linker to the symbol file, but are not otherwise processed by the linker.

7.6 Link and load addresses

The linker deals with two kinds of addresses; *link* and *load* addresses. Generally speaking the link address of a psect is the address by which it will be accessed at run time. The load address, which may or may not be the same as the link address, is the address at which the psect will start within the output file (hex or binary file etc.). In the case of the 8086 processor, the link address roughly corresponds to the offset within a segment, while the load address corresponds to the physical address of a segment. The segment address is the load address divided by 16.

Other examples of link and load addresses being different are; an initialised data psect that is copied from ROM to RAM at startup, so that it may be modified at run time; a banked text psect that is mapped from a physical (== load) address to a virtual (== link) address at run time.

The exact manner in which link and load addresses are used depends very much on the particular compiler and memory model being used.

7.7 Operation

A command to the linker takes the following form:

```
hlink1 options files ...
```

Options is zero or more linker options, each of which modifies the behaviour of the linker in some way. *Files* is one or more object files, and zero or more library names. The options recognised by the linker are listed in Table 7 - 1 on page 181 and discussed in the following paragraphs.

Table 7 - 1 Linker Options

Option	Effect
-Aclass=low-high,...	Specify address ranges for a class
-Cx	Call graph options
-Cpsect=class	Specify a class name for a global psect
-Cbaseaddr	Produce binary output file based at <i>baseaddr</i>
-Dclass=delta	Specify a class delta value
-Dsymfile	Produce old-style symbol file
-Eerrfile	Write error messages to <i>errfile</i>
-F	Produce .OBJ file with only symbol records
-Gspec	Specify calculation for segment selectors
-Hsymfile	Generate symbol file
-H+symfile	Generate enhanced symbol file
-I	Ignore undefined symbols
-Jnum	Set maximum number of errors before aborting
-K	Prevent overlaying function parameter and auto areas
-L	Preserve relocation items in .OBJ file
-LM	Preserve segment relocation items in .OBJ file
-N	Sort symbol table in map file by address order
-Nc	Sort symbol table in map file by class address order
-Ns	Sort symbol table in map file by space address order
-Mmapfile	Generate a link map in the named file
-Ooutfile	Specify name of output file
-Pspec	Specify psect addresses and ordering
-Qprocessor	Specify the processor type (for cosmetic reasons only)

1. In earlier versions of HI-TECH C the linker was called LINK.EXE

Table 7 - 1 Linker Options

Option	Effect
-S	Inhibit listing of symbols in symbol file
-Sclass=limit[,bound]	Specify address limit, and start boundary for a class of psects
-Usymbol	Pre-enter symbol in table as undefined
-Vavmap	Use file <i>avmap</i> to generate an Avocet format symbol file
-Wwarnlev	Set warning level (-10 to 10)
-Wwidth	Set map file width (>10)
-X	Remove any local symbols from the symbol file
-Z	Remove trivial local symbols from symbol file

7.7.1 Numbers in linker options

Several linker options require memory addresses or sizes to be specified. The syntax for all these is similar. By default, the number will be interpreted as a decimal value. To force interpretation as a hex number, a trailing 'H' should be added, e.g. 765FH will be treated as a hex number.

7.7.2 -Aclass=low-high,...

Normally psects are linked according to the information given to a -P option (see below) but sometimes it is desired to have a class of psects linked into more than one non-contiguous address range. This option allows a number of address ranges to be specified for a class. For example:

-ACODE=1020h-7FFEh,8000h-BFFEh

specifies that the class *CODE* is to be linked into the given address ranges. Note that a contribution to a psect from one module cannot be split, but the linker will attempt to pack each block from each module into the address ranges, starting with the first specified.

Where there are a number of identical, contiguous address ranges, they may be specified with a repeat count, e.g.

-ACODE=0-FFFFhx16

specifies that there are 16 contiguous ranges each 64k bytes in size, starting from zero. Even though the ranges are contiguous, no code will straddle a 64k boundary. The repeat count is specified as the character 'x' or '*' after a range, followed by a count.

7.7.3 -Cx

These options allow control over the call graph information which may be included in the map file produced by the linker. The -CN option removes the call graph information from the map file. The -CC option only include the critical paths of the call graph. A function call that is marked with a '*' in a full call graph is on a critical path and only these calls are included when the -CC option is used. A call graph is only produced for processors and memory models that use a compiled stack.

7.7.4 -Cpsect=class

This option will allow a psect to be associated with a specific class. Normally this is not required on the command line since classes are specified in object files.

7.7.5 -Dclass=delta

This option allows the delta value for psects that are members of the specified class to be defined. The delta value should be a number and represents the number of bytes per addressable unit of objects within the psects. Most psects do not need this option as they are defined with a delta value.

7.7.6 -Dsymfile

Use this option to produce an old-style symbol file. An old-style symbol file is an ASCII file, where each line has the link address of the symbol followed by the symbol name.

7.7.7 -Eerrfile

Error messages from the linker are written to standard error (file handle 2). Under DOS there is no convenient way to redirect this to a file (the compiler drivers will redirect standard error if standard output is redirected). This option will make the linker write all error messages to the specified file instead of the screen, which is the default standard error destination.

7.7.8 -F

Normally the linker will produce an object file that contains both program code and data bytes, and symbol information. Sometimes it is desired to produce a symbol-only object file that can be used again in a subsequent linker run to supply symbol values. The -F option will suppress data and code bytes from the output file, leaving only the symbol records.

This option can be used when producing more than one hex file for situations where the program is contained in different memory devices located at different addresses. The files for one device are compiled using this linker option to produce a symbol-only object file; this is then linked with the files for the other device. The process can then be repeated for the other files and device.

7.7.9 -Gspec

When linking programs using segmented, or bank-switched psects, there are two ways the linker can assign segment addresses, or selectors, to each segment. A segment is defined as a contiguous group of psects where each psect in sequence has both its link and load address concatenated with the previous psect in the group. The segment address or selector for the segment is the value derived when a segment type relocation is processed by the linker.

By default the segment selector will be generated by dividing the base load address of the segment by the relocation quantum of the segment, which is based on the *RELOC=* value given to psects at the assembler level. This is appropriate for 8086 real mode code, but not for protected mode or some bank-switched arrangements. In this instance the **-G** option is used to specify a method for calculating the segment selector. The argument to **-G** is a string similar to:

$A/10h-4h$

where *A* represents the load address of the segment and */* represents division. This means "Take the load address of the psect, divide by 10 hex, then subtract 4". This form can be modified by substituting *N* for *A*, * for */* (to represent multiplication), and adding rather than subtracting a constant. The token *N* is replaced by the ordinal number of the segment, which is allocated by the linker. For example:

$N*8+4$

means "take the segment number, multiply by 8 then add 4". The result is the segment selector. This particular example would allocate segment selectors in the sequence 4, 12, 20, ... for the number of segments defined. This would be appropriate when compiling for 80286 protected mode, where these selectors would represent LDT entries.

7.7.10 -Hsymfile

This option will instruct the linker to generate a symbol file. The optional argument *symfile* specifies a file to receive the symbol file. The default file name is *l.sym*.

7.7.11 -H+symfile

This option will instruct the linker to generate an *enhanced* symbol file, which provides, in addition to the standard symbol file, class names associated with each symbol and a segments section which lists each class name and the range of memory it occupies. This format is recommended if the code is to be run in conjunction with a debugger. The optional argument *symfile* specifies a file to receive the symbol file. The default file name is *l.sym*.

7.7.12 -Jerrcount

The linker will stop processing object files after a certain number of errors (other than warnings). The default number is 10, but the **-J** option allows this to be altered.

7.7.13 -K

For compilers that use a compiled stack, the linker will try and overlay function auto and parameter areas in an attempt to reduce the total amount of RAM required. For debugging purposes, this feature can be disabled with this option.

7.7.14 -I

Usually failure to resolve a reference to an undefined symbol is a fatal error. Use of this option will cause undefined symbols to be treated as warnings instead.

7.7.15 -L

When the linker produces an output file it does not usually preserve any relocation information, since the file is now absolute. In some circumstances a further "relocation" of the program will be done at load time, e.g. when running a .EXE file under DOS or a .PRG file under TOS. This requires that some information about what addresses require relocation is preserved in the object (and subsequently the executable) file. The -L option will generate in the output file one null relocation record for each relocation record in the input.

7.7.16 -LM

Similar to the above option, this preserves relocation records in the output file, but only segment relocations. This is used particularly for generating .EXE files to run under DOS.

7.7.17 -Mmapfile

This option causes the linker to generate a link map in the named file, or on the standard output if the file name is omitted. The format of the map file is illustrated in Section 7.9 on page 189.

7.7.18 -N, -Ns and -Nc

By default the symbol table in the link map will be sorted by name. The -N option will cause it to be sorted numerically, based on the value of the symbol. The -Ns and -Nc options work similarly except that the symbols are grouped by either their Space type, or Class type.

7.7.19 -Ooutfile

This option allows specification of an output file name for the linker. The default output file name is *l.obj*. Use of this option will override the default.

7.7.20 -Pspec

Psects are linked together and assigned addresses based on information supplied to the linker via -P options. The argument to the -P option consists basically of comma separated sequences thus:

```
-Ppsect=lnkaddr+min/ldaddr+min,psect=lnkaddr/ldaddr, ...
```

There are several variations, but essentially each psect is listed with its desired link and load addresses, and a minimum value. All values may be omitted, in which case a default will apply, depending on previous values.

The minimum value (*min*) is preceded by a + sign, if present. It sets a minimum value for the link or load address. The address will be calculated as described below, but if it is less than the minimum then it will be set equal to the minimum.

The link and load addresses are either numbers as described above, or the names of other psects or classes, or special tokens. If a link address is omitted, the psect's link address will be derived from the top of the previous psect, e.g.

```
-Ptext=100h,data,bss
```

In this example the text psect is linked at 100 hex (its load address defaults to the same). The data psect will be linked (and loaded) at an address which is 100 hex plus the length of the text psect, rounded up as necessary if the data psect has a *RELOC*= value associated with it. Similarly, the bss psect will concatenate with the data psect.

If the load address is omitted entirely, it defaults to the same as the link address. If the slash (/) character is supplied, but no address is supplied after it, the load address will concatenate with the previous psect, e.g.

```
-Ptext=0,data=0/,bss
```

will cause both text and data to have a link address of zero, text will have a load address of 0, and data will have a load address starting after the end of text. The bss psect will concatenate with data for both link and load addresses.

The load address may be replaced with a dot (.) character. This tells the linker to set the load address of this psect to the same as its link address. The link or load address may also be the name of another (already linked) psect. This will explicitly concatenate the current psect with the previously specified psect, e.g.

```
-Ptext=0,data=8000h/,bss/. -Pnvram=bss,heap
```

This example shows text at zero, data linked at 8000h but loaded after text, bss is linked and loaded at 8000h plus the size of data, and nvram and heap are concatenated with bss. Note here the use of two -P options. Multiple -P options are processed in order.

If -A options have been used to specify address ranges for a class then this class name may be used in place of a link or load address, and space will be found in one of the address ranges. For example:

```
-ACODE=8000h-BFFeh, E000h-FFFEh
-Pdata=C000h/CODE
```

This will link data at C000h, but find space to load it in the address ranges associated with CODE. If no sufficiently large space is available, an error will result. Note that in this case the data psect will still be assembled into one contiguous block, whereas other psects in the class CODE will be distributed into the address ranges wherever they will fit. This means that if there are two or more psects in class CODE, they may be intermixed in the address ranges.

Any psects allocated by a -P option will have their load address range subtracted from any address ranges specified with the -A option. This allows a range to be specified with the -A option without knowing in advance how much of the lower part of the range, for example, will be required for other psects.

7.7.21 -Qprocessor

This option allows a processor type to be specified. This is purely for information placed in the map file. The argument to this option is a string describing the processor.

7.7.22 -S

This option prevents symbol information relating from being included in the symbol file produced by the linker. Segment information is still included.

7.7.23 -Sclass=limit[, bound]

A class of psects may have an upper address *limit* associated with it. The following example places a limit on the maximum address of the CODE class of psects to one less than 400h.

```
-SCODE=400h
```

Note that to set an upper limit to a psect, this must be set in assembler code (with a *LIMIT=* flag on a psect directive).

If the *bound* (boundary) argument is used, the class of psects will start on a multiple of the bound address. This example places the FARCODE class of psects at a multiple of 1000h, but with an upper address limit of 6000h:

```
-SFARCODE=6000h, 1000h
```

7.7.24 -Usymbol

This option will enter the specified symbol into the linker's symbol table as an undefined symbol. This is useful for linking entirely from libraries, or for linking a module from a library where the ordering has been arranged so that by default a later module will be linked.

7.7.25 -Vavmap

To produce an Avocet format symbol file, the linker needs to be given a map file to allow it to map psect names to Avocet memory identifiers. The avmap file will normally be supplied with the compiler, or created automatically by the compiler driver as required.

7.7.26 -Wnum

The -W option can be used to set the warning level, in the range -9 to 9, or the width of the map file, for values ≥ 10 .

-W9 will suppress all warning messages. -W0 is the default. Setting the warning level to -9 (-W-9) will give the most comprehensive warning messages.

7.7.27 -X

Local symbols can be suppressed from a symbol file with this option. Global symbols will always appear in the symbol file.

7.7.28 -Z

Some local symbols are compiler generated and not of interest in debugging. This option will suppress from the symbol file all local symbols that have the form of a single alphabetic character, followed by a digit string. The set of letters that can start a trivial symbol is currently "klfLSu". The -Z option will strip any local symbols starting with one of these letters, and followed by a digit string.

7.8 Invoking the Linker

The linker is called HLINK, and normally resides in the BIN subdirectory of the compiler installation directory. It may be invoked with no arguments, in which case it will prompt for input from standard input. If the standard input is a file, no prompts will be printed. This manner of invocation is generally useful if the number of arguments to HLINK is large. Even if the list of files is too long to fit on one line, continuation lines may be included by leaving a backslash ('\') at the end of the preceding line. In this fashion, HLINK commands of almost unlimited length may be issued. For example a link command file called X.LNK and containing the following text:

```
-Z -OX.OBJ -MX.MAP \  
-Ptext=0,data=0/,bss,nvram=bss/. \  
X.OBJ Y.OBJ Z.OBJ C:\HT-Z80\LIB\Z80-SC.LIB
```

may be passed to the linker by one of the following:

```
hlink @x.lnk  
hlink <x.lnk
```

7.9 Map Files

The map file contains information relating to the relocation of psects and the addresses assigned to symbols within those psects. The sections in the map file are as follows; first is a copy of the command line used to invoke the linker. This is followed by the version number of the object code in the first file linked, and the machine type. This is optionally followed by call graph information, depended on the processor and memory model selected. Then are listed all object files that were linked, along with their psect information. Libraries are listed, with each module within the library. The TOTALS section summarises the psects from the object files. The SEGMENTS section summarises major memory groupings. This will typically show RAM and ROM usage. The segment names are derived from the name of the first psect in the segment.

Lastly (not shown in the example) is a symbol table, where each global symbol is listed with its associated psect and link address.

Linker command line:

```
-z -Mmap -pvectors=00h,text,strings,const,im2vecs -pbaseram=00h \
  -pramstart=08000h,data/im2vecs,bss/. ,stack=09000h -pnvram=bss,heap \
  -oC:\TEMP\1.obj C:\HT-Z80\LIB\rtz80-s.obj hello.obj \
  C:\HT-Z80\LIB\z80-sc.lib
```

Object code version is 2.4

Machine type is Z80

	Name	Link	Load	Length	Selector
C:\HT-Z80\LIB\rtz80-s.obj	vectors	0	0	71	
	bss	8000	8000	24	
	const	FB	FB	1	0
	text	72	72	82	
hello.obj	text	F4	F4	7	

C:\HT-Z80\LIB\z80-sc.lib

powerup.obj	vectors	71	71	1	
-------------	---------	----	----	---	--

TOTAL	Name	Link	Load	Length
CLASS	CODE			
	vectors	0	0	72
	const	FB	FB	1
	text	72	72	89
CLASS	DATA			
	bss	8000	8000	24

SEGMENTS	Name	Load	Length	Top	Selector
	vectors	000000	0000FC	0000FC	0

bss 008000 000024 008024 8000

7.9.1 Call Graph Information

A call graph is produced for chip types and memory models that use a compiled stack, rather than a hardware stack, to facilitate parameter passing between functions and auto variables defined within a function. When a compiled stack is used, functions are not re-entrant since the function will use a fixed area of memory for its local objects (parameters/auto variables). A function called `foo`, for example, will use symbols like `?_foo` for parameters and `?a_foo` for auto variables. Compilers such as the PIC, 6805 and V8 use compiled stacks. The 8051 compiler uses a compiled stack in small and medium memory models. The call graph shows information relating to the placement of function parameters and auto variables by the linker. A typical call graph may look something like:

Call graph:

```
*_main size 0,0 offset 0
    _init size 2,3 offset 0
        _ports size 2,2 offset 5
*    _sprintf size 5,10 offset 0
*    _putch
        INDIRECT 4194
            INDIRECT 4194
                _function_2 size 2,2 offset 0
                _function size 2,2 offset 5
*_isr->_incr size 2,0 offset 15
```

The graph shows the functions called and the memory usage (RAM) of the functions for their own local objects. In the example above, the symbol `_main` is associated with the function `main`. It is shown at the far left of the call graph. This indicates that it is the root of a call tree. The run-time code has the `FNROOT` assembler directive that specifies this. The size field after the name indicates the number of parameters and auto variables, respectively. Here, `main()` takes no parameters and defines no auto variables. The offset field is the offset at which the function's parameters and auto variables have been placed from the beginning of the area of memory used for this purpose. The run-time code contains a `FNCONF` directive which tells the compiler in which psect parameters and auto variables should reside. This memory will be shown in the map file under the name `COMMON`.

`Main()` calls a function called `init`. This function uses a total of two bytes of parameters (it may be two chars or one int; that is not important) and has three bytes of auto variables. These figures are the total of bytes of *memory* consumed by the function. If the function was passed a two-byte int, but that

was done via a register, then the two bytes would not be included in this total. Since `main()` did not use any of the local object memory, the offset of `init()`'s memory is still at 0.

The function `init` itself calls another function called `ports`. This function uses two bytes of parameters and another two bytes of auto variables. Since `ports()` is called by `init()`, its local variables cannot be overlapped with those of `init()`'s, so the offset is 5, which means that `ports()`'s local objects were placed immediately after those of `init()`'s.

The function `main` also calls `sprintf()`. Since the function `sprintf` is not active at the same time as `init()` or `ports()`, their local objects can be overlapped and the offset is hence set to 0. `Sprintf()` calls a function `putch`, but this function uses no memory for parameters (the char passed as argument is apparently done so via a register) or locals, so the size and offset are zero and are not printed.

`Main()` also calls another function indirectly using a function pointer. This is indicated by the two `INDIRECT` entries in the graph. The number following is the signature value of functions that could potentially be called by the indirect call. This number is calculated from the parameters and return type of the functions the pointer can indirectly call. The names of any functions that have this signature value are listed underneath the `INDIRECT` entries. Their inclusion does not mean that they were called (there is no way to determine that), but that they could potentially be called.

The last line shows another function whose name is at the far left of the call graph. This implies that this is the root of another call graph tree. This is an interrupt function which is not called by any code, but which is automatically invoked when an enabled interrupt occurs. This interrupt routine calls the function `incr()`, which is shown shorthand in the graph by the “->” symbol followed by the called function's name instead of having that function shown indented on the following line. This is done whenever the calling function does not take parameters, nor defines any variables.

Those lines in the graph which are starred (*) are those functions which are on a critical path in terms of RAM usage. For example, in the above, (`main()` is a trivial example) consider the function `sprintf`. This uses a large amount of local memory and if you could somehow rewrite it so that it used less local memory, it would reduce the entire program's RAM usage. The functions `init` and `ports` have had their local memory overlapped with that of `sprintf()`, so reducing the size of these functions' local memory will have no effect on the program's RAM usage. Their memory usage could be increased, as long as the total size of the memory used by these two functions did not exceed that of `sprintf()`, with no additional memory used by the program. So if you have to reduce the amount of RAM used by the program, look at those functions that are starred.

If, when searching a call graph, you notice that a function's parameter and auto areas have been overlapped (i.e. `?a_foo` was placed at the same address as `?_foo`, for example), then check to make sure that you have actually called the function in your program. If the linker has not seen a function actually called, then it overlaps these areas of memory since that area is not needed. This is a consequence of the linker's ability to overlap the local memory areas of functions which are not active at the same

time. Once the function is called, unique addresses will be assigned to both the parameters and auto objects.

If you are writing a routine that calls C code from assembler, you will need to include the appropriate assembler directives to ensure that the linker sees the C function being called.

7.10 Librarian

The librarian program, LIBR, has the function of combining several object files into a single file known as a library. The purposes of combining several such object modules are several.

- ☐ fewer files to link
- ☐ faster access
- ☐ uses less disk space

In order to make the library concept useful, it is necessary for the linker to treat modules in a library differently from object files. If an object file is specified to the linker, it will be linked into the final linked module. A module in a library, however, will only be linked in if it defines one or more symbols previously known, but not defined, to the linker. Thus modules in a library will be linked only if required. Since the choice of modules to link is made on the first pass of the linker, and the library is searched in a linear fashion, it is possible to order the modules in a library to produce special effects when linking. More will be said about this later.

7.10.1 The Library Format

The modules in a library are basically just concatenated, but at the beginning of a library is maintained a directory of the modules and symbols in the library. Since this directory is smaller than the sum of the modules, the linker is speeded up when searching a library since it need read only the directory and not all the modules on the first pass. On the second pass it need read only those modules which are required, seeking over the others. This all minimises disk I/O when linking.

It should be noted that the library format is geared exclusively toward object modules, and is not a general purpose archiving mechanism as is used by some other compiler systems. This has the advantage that the format may be optimized toward speeding up the linkage process.

7.10.2 Using the Librarian

The librarian program is called LIBR, and the format of commands to it is as follows:

```
libr options k file.lib file.obj ...
```

Interpreting this, LIBR is the name of the program, *options* is zero or more librarian options which affect the output of the program. *k* is a key letter denoting the function requested of the librarian (replacing,

extracting or deleting modules, listing modules or symbols), *file.lib* is the name of the library file to be operated on, and *file.obj* is zero or more object file names.

The librarian options are listed in Table 7 - 2.

Table 7 - 2 Librarian Options

Option	Effect
-Pwidth	specify page width
-W	suppress non-fatal errors

The key letters are listed in Table 7 - 3.

Table 7 - 3 Librarian Key Letter Commands

Key	Meaning
r	Replace modules
d	Delete modules
x	Extract modules
m	List modules
s	List modules with symbols

When replacing or extracting modules, the *file.obj* arguments are the names of the modules to be replaced or extracted. If no such arguments are supplied, all the modules in the library will be replaced or extracted respectively. Adding a file to a library is performed by requesting the librarian to replace it in the library. Since it is not present, the module will be appended to the library. If the *r* key is used and the library does not exist, it will be created.

Under the *d* key letter, the named object files will be deleted from the library. In this instance, it is an error not to give any object file names.

The *m* and *s* key letters will list the named modules and, in the case of the *s* keyletter, the symbols defined or referenced within (global symbols only are handled by the librarian). As with the *r* and *x* key letters, an empty list of modules means all the modules in the library.

7.10.3 Examples

Here are some examples of usage of the librarian. The following lists the global symbols in the modules a.obj, b.obj and c.obj:

```
LIBR s file.lib a.obj b.obj c.obj
```

This command deletes the object modules a.obj, b.obj and 2.obj from the library file.lib:

```
LIBR d file.lib a.obj b.obj 2.obj
```

7.10.4 Supplying Arguments

Since it is often necessary to supply many object file arguments to LIBR, and command lines are restricted to 127 characters by CP/M and MS-DOS, LIBR will accept commands from standard input if no command line arguments are given. If the standard input is attached to the console, LIBR will prompt for input. Multiple line input may be given by using a backslash as a continuation character on the end of a line. If standard input is redirected from a file, LIBR will take input from the file, without prompting. For example:

```
libr
libr> r file.lib 1.obj 2.obj 3.obj \
libr> 4.obj 5.obj 6.obj
```

will perform much the same as if the .obj files had been typed on the command line. The libr> prompts were printed by LIBR itself, the remainder of the text was typed as input.

```
libr <lib.cmd
```

Libr will read input from lib.cmd, and execute the command found therein. This allows a virtually unlimited length command to be given to LIBR.

7.10.5 Listing Format

A request to LIBR to list module names will simply produce a list of names, one per line, on standard output. The s keyletter will produce the same, with a list of symbols after each module name. Each symbol will be preceded by the letter D or U, representing a definition or reference to the symbol respectively. The -P option may be used to determine the width of the paper for this operation. For example LIBR -P80 s file.lib will list all modules in file.lib with their global symbols, with the output formatted for an 80 column printer or display.

7.10.6 Ordering of Libraries

The librarian creates libraries with the modules in the order in which they were given on the command line. When updating a library the order of the modules is preserved. Any new modules added to a library after it has been created will be appended to the end.

The ordering of the modules in a library is significant to the linker. If a library contains a module which references a symbol defined in another module in the same library, the module defining the symbol should come after the module referencing the symbol.

7.10.7 Error Messages

LIBR issues various error messages, most of which represent a fatal error, while some represent a harmless occurrence which will nonetheless be reported unless the -W option was used. In this case all warning messages will be suppressed.

7.11 Objtohex

The HI-TECH linker is capable of producing simple binary files, or object files as output. Any other format required must be produced by running the utility program OBJTOHEX. This allows conversion of object files as produced by the linker into a variety of different formats, including various hex formats. The program is invoked thus:

```
objtohex options inputfile outputfile
```

All of the arguments are optional. If *outputfile* is omitted it defaults to *l.hex* or *l.bin* depending on whether the -b option is used. The *inputfile* defaults to *l.obj*.

The options for OBJTOHEX are listed in Table 7 - 4 on page 196.: Where an address is required, the format is the same as for HLINK.

7.11.1 Checksum Specifications

The checksum specification allows automated checksum calculation. The checksum specification takes the form of several lines, each line describing one checksum. The syntax of a checksum line is:

```
addr1-addr2 where1-where2 +offset
```

All of *addr1*, *addr2*, *where1*, *where2* and *offset* are hex numbers, without the usual H suffix. Such a specification says that the bytes at *addr1* through to *addr2* inclusive should be summed and the sum placed in the locations *where1* through *where2* inclusive. For an 8 bit checksum these two addresses should be the same. For a checksum stored low byte first, *where1* should be less than *where2*, and vice versa. The +*offset* is optional, but if supplied, the value *offset* will be used to initialise the checksum. Otherwise it is initialised to zero. For example:

```
0005-1FFF 3-4 +1FFF
```

This will sum the bytes in 5 through 1FFFH inclusive, then add 1FFFH to the sum. The 16 bit checksum will be placed in locations 3 and 4, low byte in 3. The checksum is initialised with 1FFFH to provide protection against an all zero rom, or a rom misplaced in memory. A run time check of this checksum would add the last address of the rom being checksummed into the checksum. For the rom in question, this should be 1FFFH. The initialization value may, however, be used in any desired fashion.

Table 7 - 4 Objtohex Options

Option	Meaning
-A	Produce an ATDOS .ATX output file
-Bbase	Produce a binary file with offset of <i>base</i> . Default file name is <i>l.obj</i>
-Cckfile	Read a list of checksum specifications from <i>ckfile</i> or standard input
-D	Produce a .COD file
-E	Produce an MS-DOS .EXE file
-Ffill	Fill unused memory with bytes of value <i>fill</i> - default value is 0FFh
-I	Produce an Intel HEX file with linear addressed extended records.
-L	Pass relocation information into the output file (used with .EXE files)
-M	Produce a Motorola HEX file (S19, S28 or S37 format)
-N	Produce an output file for Minix
-Pstk	Produce an output file for an Atari ST, with optional stack size
-R	Include relocation information in the output file
-Sfile	Write a symbol file into <i>file</i>
-T	Produce a Tektronix HEX file. -TE produces an extended TekHEX file.
-U	Produce a COFF output file
-UB	Produce a UBROF format file
-V	Reverse the order of words and long words in the output file
-x	Create an x.out format file

7.12 Cref

The cross reference list utility CREF is used to format raw cross-reference information produced by the compiler or the assembler into a sorted listing. A raw cross-reference file is produced with the -CR option to the compiler. The assembler will generate a raw cross-reference file with a -C option (most assemblers) or by using an OPT CRE directive (6800 series assemblers) or a XREF control line (PIC assembler). The general form of the CREF command is:

```
cref options files
```

where *options* is zero or more options as described below and *files* is one or more raw cross-reference files. CREF takes the options listed in Table 7 - 5 on page 197. Each option is described in more detail in the following paragraphs.

Table 7 - 5 Cref Options

Option	Meaning
-Fprefix	Exclude symbols from files with a pathname or filename starting with <i>prefix</i>
-Hheading	Specify a heading for the listing file
-Llen	Specify the page length for the listing file
-Ooutfile	Specify the name of the listing file
-Pwidth	Set the listing width
-Sstoplist	Read file <i>stoplist</i> and ignore any symbols listed.
-Xprefix	Exclude any symbols starting with the given <i>prefix</i>

7.12.1 -Fprefix

It is often desired to exclude from the cross-reference listing any symbols defined in a system header file, e.g. <stdio.h>. The -F option allows specification of a path name prefix that will be used to exclude any symbols defined in a file whose path name begins with that prefix. For example, -F\ will exclude any symbols from all files with a path name starting with \.

7.12.2 -Hheading

The -H option takes a string as an argument which will be used as a header in the listing. The default heading is the name of the first raw cross-ref information file specified.

7.12.3 -Llen

Specify the length of the paper on which the listing is to be produced, e.g. if the listing is to be printed on 55 line paper you would use a -L55 option. The default is 66 lines.

7.12.4 -Ooutfile

Allows specification of the output file name. By default the listing will be written to the standard output and may be redirected in the usual manner. Alternatively *outfile* may be specified as the output file name.

7.12.5 -Pwidth

This option allows the specification of the width to which the listing is to be formatted, e.g. -P132 will format the listing for a 132 column printer. The default is 80 columns.

7.12.6 -Sstoplist

The -S option should have as its argument the name of a file containing a list of symbols not to be listed in the cross-reference. Multiple stoplists may be supplied with multiple -S options.

7.12.7 -Xprefix

The -X option allows the exclusion of symbols from the listing, based on a prefix given as argument to -X. For example if it was desired to exclude all symbols starting with the character sequence *xyz* then the option -X*xyz* would be used. If a digit appears in the character sequence then this will match any digit in the symbol, e.g. -XX0 would exclude any symbols starting with the letter *X* followed by a digit.

CREF will accept wildcard filenames and I/O redirection. Long command lines may be supplied by invoking CREF with no arguments and typing the command line in response to the *cref*> prompt. A backslash at the end of the line will be interpreted to mean that more command lines follow.

7.13 Memmap

MEMMAP has been individualized for each processor. The MEMMAP program that appears in your \bin directory will conform with the following criteria; memmapXX.exe where *XX* stands for the processor type. From here on, we will be referring to memmapXX.exe as MEMMAP, as to cover all processors.

At the end of compilation and linking, HPD and the command line compiler produce a summary of memory usage. If, however, the compilation is performed in separate stages and the linker is invoked explicitly, this memory information is not displayed. The MEMMAP program reads the information stored in the map file and produces either a summary of psect address allocation or a memory map of program sections similar to that shown by HPD and the command line compiler.

7.13.1 Using MEMMAP

A command to the memory usage program takes the form:

```
memmap options file
```

Options is zero or more MEMMAP options which are listed in Table 7 - 6 on page 198. *File* is the name

Table 7 - 6 Memmap options

Option	Effect
-P	Print psect usage map
-Wwid	Specifies width to which address are printed

of a map file. Only one map file can be processed by MEMMAP.

7.13.1.1 -P

The default behaviour of MEMMAP is to produce a segment memory map. This output is similar to that printed by HPD and the command line compiler after compilation and linking. This behaviour can be

changed by using the -P option. This forces a psect usage map to be printed. The output in this case will be similar to that shown by the HPD's **Memory Usage Map** item under the **Utility** menu or if the -PSECTMAP option is used with the command line compiler.

7.13.1.2 -Wwid

The width to which addresses are printed can be adjusted by using the -W option. The default width is determined in respect to the processor's address range. Depending on the type of processor used, determines the default width of the printed address, for example a processor with less than or equal to 64k will have a default width of 4. Whereas a processor with greater than 64k may have a default value of 6 digits.

Index

Symbols

- ! macro quote character 174
- #asm directive 127
- #define 95
- #endasm directive 127
- #pragma directives 138
- #undef 103
- \$ assembler label character 157
- \$ location counter symbol 158
- % macro argument prefix 174
- . psect address symbol 185
- ... symbol 130
- .as files 84
- .c files 84
- .cmd files 92, 194
- .crf files 95, 154
- .hex files 99, 100, 103
- .lib files 84, 105, 192, 193
- .lnk files 45, 188
- .lst files 81, 94
- .map files 83
- .obj files 84, 185, 193
- .opt files 153
- .pre files 79
- .prj files 83, 88, 141
- .pro files 101
- .rlf files 27
- .sdb files 25
- .sym files 106, 184, 187
- .ubr files 104
- / psect address symbol 185
- :: macro comment suppresser 171

- <<>>menu 59, 73
 - About HTLPIC 73
 - Setup... 73
- <> macro argument list 174
- ? assembler special label character 157
- ??nnnn type symbols 158, 171
- ?_xxxx type symbols 131, 190
- ?a_xxxx type symbols 134, 168, 190
- @ address construct 115, 129
- @ in a macro 171
- _ assembler special label character 157
- _Bxxxx type symbols 37, 138
- _CONFIG macro 107
- _Hxxxx type symbols 35, 138
- _IDLOC macro 108
- _Lxxxx type symbols 35, 138
- _READ_OSCCAL_DATA macro 110
- _xxxx type symbols 136

Numerics

- 24-bit floating point format 114
- 32-bit floating point format 114

A

- abs psect flag 164, 165
- abs psect flags 34
- absolute addresses 166
- absolute object files 185
- absolute psects 162, 164, 165
- absolute variables 43, 115, 140
 - bits 112
 - structures 116

- addresses
 - link 180, 185
 - link addresses
 - load 32
 - load 180, 185
 - unresolved in listing file 26
- addressing unit size 164
- align directive 172
- alignment
 - psects 165
 - within psects 172
- ANSI standard
 - conformance 103
 - disabling 81
 - divergence from 105
 - implementation-defined behaviour 121
- argument area 130
 - size of 169
- argument passing 130
- ASCII characters 113
- ASCII table 87
- asm() C directive 127
- ASPIC
 - controls 174
 - table of 175
 - directives 162
 - processor 125
 - table of 163
 - expressions 160
 - generated symbols 158
 - labels 158
 - location counter 158
 - numbers and bases 156
 - operators, table of 161
 - options 153
 - options, table of 154
 - psect flags
 - abs 34
 - global 33
 - local 33
 - ovlrd 33
 - reloc 34
 - special characters 157
 - statements 160
 - symbols 158
- ASPIC controls
 - cond 175
 - expand 175
 - include 175
 - list 175
 - nocond 176
 - noexpand 176
 - nolist 176
 - noxref 176
 - page 176
 - space 176
 - subtitle 176
 - table of 175
 - title 176
 - xref 177
- ASPIC directives
 - align 172
 - db 160, 166
 - defl 166
 - ds 167
 - dw 167
 - else 169
 - elsif 169
 - end 164
 - endif 169
 - endm 170
 - equ 166
 - fnaddr 167
 - fnarg 167
 - fnbreak 168
 - fncall 168

-
- fnconf 168
 - fnindir 168
 - fnroot 169
 - fnsiz 125, 169
 - global 125, 162
 - if 169
 - irp 172
 - irpc 172
 - local 158, 171
 - macro 170
 - org 166
 - pagesel 174
 - processor 174
 - psect 30, 125, 160, 164
 - psect flags
 - abs 164, 165
 - bit 164
 - class 164
 - delta 165
 - global 165
 - limit 165
 - local 165
 - ovrld 165
 - pure 165
 - reloc 165
 - size 165
 - space 165
 - with 165
 - rept 172
 - set 166
 - signat 174
 - signat directive 128
 - ASPIC options
 - A 153
 - C 154
 - Cchipinfo 154
 - E 154
 - Flength 154
 - H 155
 - I 155
 - Llistfile 155
 - O 155
 - Ooutfile 155
 - processor 153
 - Raddress 155
 - S 155
 - table of 154
 - U 155
 - V 155
 - Wwidth 155
 - X 155
 - assembler 25, 153
 - accessing C objects 126
 - base specifiers 156
 - character set 156
 - command line options 153
 - conditional 170
 - constants 156
 - character 157
 - double 156
 - float 156
 - hexadecimal 156
 - controls 174
 - table of 175
 - default radix 156
 - delimiters 157
 - destination operand 156
 - differences between Microchip 156
 - directives 162
 - expressions 160
 - generating from C 103
 - identifiers 157, 158
 - significance of 157
 - include files 175
 - initializing
 - bytes 166
 - words 167
 - in-line 126

- label field 160
- labels 158
 - numeric 159
 - symbolic 158
- line numbers 155
- macros 173
- mixing with C 125
- operators, table of 161
- optimizer 25, 153, 155
- options, table of 154
- pseudo-ops 162
- radix specifiers 156
- repeating macros 172
- reserving
 - bytes 166
 - locations 167
 - words 167
- special characters 157
- strings 160
- symbols 162
- user-defined symbols 157

- assembler code 155
 - called by C 125
- assembler errors
 - suppressing 155
- assembler files 20
 - preprocessing 81, 101
 - using hexadecimal constants 155
- assembler labels 158
- assembler listings 26, 81, 94, 155
 - blank lines 176
 - disabling macro expansion 176
 - excluding conditional code 176
 - expanding macros 175
 - including conditional code 175
 - new page 176
 - page length 154
 - page width 155
 - subtitle 176

- title 176
 - turning off 175
 - turning on 175
- assembler options 153
- auto indent mode 67
- auto variable area 130
- auto variables 133
 - bank location 119
 - symbol names 168
- auto-repeat rate, mouse 73
- autosave 74

B

- ballistic threshold, mouse 73
- bank1 qualifier 118, 119
- bank2 qualifier 118, 119
- bank3 qualifier 118, 119
- banks
 - chipinfo file 129
 - RAM banks 119
- bases
 - assembler 156
 - C source 111
- batch files 92, 97
- begin block 69
- biased exponent 115
- binary constants
 - assembler 157
 - C 111
- binary files 94, 129
- bit
 - keyword 112
 - psect flag 164
- bit fields 116
- bit instructions 108
- bit types 112, 116, 136, 164
 - absolute 112

bit-addressable Registers 112

block commands 69

begin 69

comment/uncomment 71

copy 70

delete 70

end 70

go to end 70

go to start 70

hide/display 70

indent 71

key equivalents
table of 69

move 70

outdent 71

read 70

write 71

block hide 71

boolean types 111

bss psect 180

clearing 180

button

continue 65

fix 65

help 65

hide 65

next 67

previous 67

search 67

buttons 59

C

C mode 67

C source listings 21, 81

example of 21

calculator 86

calibration constant 110

call graph 124, 168, 169, 190

char types 113

signed 103

chicken sheds 158

chipinfo files 154

class psect flag 164

classes 183

address ranges 182

boundary argument 187

upper address limit 187

clear clipboard 72

clearing bits 109

clipboard 69, 71

clear 76

copy 74

cut 74

delete selection 75

hide 74

paste 74

selecting text 71

show 76

clipboard commands 72

clear 72

copy 70, 72

cut 72

delete 70, 72

hide 72

paste 72

show 72

clist utility 21

clock, enabling 59

clrtext psect 135

clutches 42

code generator 24

code protection fuses 107

colours 53

attributes, table of 54

settings 55

values, table of 54

command files

PICL 92

- command line driver 91
- command lines
 - HLINK, long command lines 188
 - long 91, 194
 - verbose option 104
- commands
 - block 69
 - clipboard 72
 - HTLPIC keyboard 67
- comment block 71
- commenting code 76
- comments
 - block 71
 - C++ style 71, 76
- compile menu 63, 78
 - compile and link 78
 - compile to .AS 79
 - compile to .OBJ 79
 - disable non-ANSI features 81
 - generate assembler listing 81
 - generate C source listing 81
 - identifier length 80
 - optimization 80
 - preprocess only to .PRE 79
 - stop on warnings 79
 - warning level 80
- compiled stack 190
- compiler
 - options 92
 - options help 90
 - overview 17
 - release notes 90
 - technical support 89
- compiler errors 97
 - format 96
- compiler generated psects 134
- compiling
 - to assembler file 79, 103
 - to executable file 78
 - to object file 79, 94
 - to preprocessor file 79
- cond assembler control 174
- conditional assembly 169
- config psect 135
- configuration
 - fuses 107
 - word 135
- console I/O functions 142
- const
 - keyword 117
 - pointers 117, 120, 121
- constants
 - assembler 156
 - C specifiers 111
 - placement of 118
- constn psect 118, 135
- context retrieval 123
- context saving 123
 - in-line assembly 141
 - midrange 123
- continue button 65
- copy 72
- copy block 70
- copyright notice 103
- cpp application 21
- creating
 - libraries 193
 - projects 83
 - source files 73
- CREF 154, 196
 - command line arguments 196
 - options 197
 - Fprefix 197
 - Hheading 197
 - Llen 197
 - Ooutfile 197
 - Pwidth 197

- Sstoplist 197
- Xprefix 198
- cromwell application 29
- cross reference
 - disabling 176
 - enabling 177
 - generating 196
 - generation 154
 - list utility 196
- cross reference listings 95
 - excluding header symbols 197
 - excluding symbols 197, 198
 - headers 197
 - output name 197
 - page length 197
 - page width 197
- cstrings psect 135
- ctextn psect 134
- cut 72

D

- data psect 180
 - copying 180
- data psects 195
- data types 110
 - 16-bit integer 113
 - 32-bit integer 113
 - 8-bit integer 113
 - bit 112, 116, 136, 164
 - char 113
 - floating point 114
 - int 113
 - long 113
 - short 113
 - table of 111
- db directive 160, 166
- debug information 25, 78, 98, 106, 155

- default libraries 92
- default psect 162
- defl directive 166
- delete block 70
- delete selection 72
- delta psect flag 45, 165, 183
- dependency information 27, 78
- directives
 - asm, C 127
 - assembler 162
 - table of 163
- DOS
 - command line limitations 52, 91
 - commands 85
 - defining commands 87
 - free memory 73
 - shell 85
- double type
 - size of 95
- ds 167
- DUMP 26
- dw 167

E

- edit menu 74
 - C colour coding 76
 - comment/uncomment 76
 - copy 74
 - cut 74
 - delete selection 75
 - go to line 76
 - hide 74
 - indent 76
 - outdent 76
 - paste 74
 - replace 75
 - search 75

set tab size 76
show clipboard 76
edit window 66
editor 65
 auto indent mode 67
 autosave 74
 begin block 69
 block commands 69
 block hide/display 70
 C mode 67
 clear clipboard 72
 clipboard 69, 71
 colours 53
 attributes, table of 54
 settings, table of 55
 values, table of 54
 comment/uncomment block 71
 commenting/uncommenting code 76
 content region 66
 copy 72, 74
 copy block 70
 cut 72, 74
 delete block 70
 delete selection 72, 75
 end block 70
 frame area 66
 go to block end 70
 go to block start 70
 go to line 76
 hide 72, 74
 hide block 71
 indent
 block 71
 mode 67
 indenting 76
 insert mode 66
 keyboard commands 67
 keys
 help 89
 table of 68, 69
 move block 70
 new 73
 opening recent files 74
 outdent block 71
 outdenting 76
 overwrite mode 66
 paste 72, 74
 read block 70
 replace text 75
 save 74
 search 67, 75
 selecting text 71
 show clipboard 72, 76
 status line 66
 syntax highlighting 76
 tab size 76
 write block 71
 zoom command 67
ellipsis symbol 130
else directive 169
elsif directive 169
end block 70
end directive 164
end_init psect 135
endif directive 169
endm directive 170
enhanced S-Record 92
enhanced symbol files 184
environment variable
 HTC_ERR_FORMAT 96
 HTC_WARN_FORMAT 96
 TEMP 52
equ directive 160, 166
equating symbols 166
error files 97
 creating 183

error messages
 formatting 96
 LIBR 195
 used by HTLPIC 154

errors
 auto-fix 65
 format 96
 redirecting 97
expand assembler control 174
exponent 114
expressions
 assembler 160
 relocatable 160
extern keyword 125
external ROM
 HTLPIC 78

F

fastcall functions 133, 134
fcall mnemonic 155, 156, 174
file formats 17, 77
 American Automation hex 92
 assembler 84
 assembler listing 81, 94
 binary 94
 C source 84
 C source listings 81
 command 92, 194
 cross reference 154, 196
 cross reference listings 95
 DOS executable 185
 enhanced Motorola S-Record 92
 enhanced symbol 184
 Intel hex 99
 library 84, 105, 192, 193
 link 188
 map 83, 189

 Motorola hex 100
 object 84, 94, 185, 193
 optimizer 153
 preprocessor 79, 101
 project 83, 88, 141
 project files 83
 prototype 101
 relocatable listing file 27
 specifying 100, 155
 S-Record files 100
 symbol 184
 symbol files 106
 symbolic debug 25
 Tektronix hex 103
 TOS executable 185
 UBROF 104
file menu 73
 autosave 74
 clear pick list 74
 new 73
 open 73
 pick list 74
 quit 74
 save 74
 save as 74
fix button 65
fixing errors 65
fixup 28
flags
 psect 164
float type
 size of 95
float_text psect 135
float24 pseudo-function 157
floating point data types 77, 114
 24-bit format 95, 114
 32-bit format 95, 114
 biased exponent 115

- exponent 115
- format 114
- format examples 114
- mantissa 114
- floating point operations 135
- floating point routines, fast 98
- fnaddr directive 167
- fnarg directive 167
- fnbreak directive 168
- fnconf directive 168
- fnindir directive 168
- fnroot directive 169, 190
- fnsz directive 169
- function
 - return values 131
 - 16-bit 131
 - 32-bit 131
 - 8-bit 131
 - structures 132
- function calls 168
 - indirect 168
- function parameters 119, 131
- function pointers 120
- function prototypes 128, 174
 - ellipsis 130
- function return values 131
- functions
 - argument area 131
 - argument passing 130
 - calling conventions 133
 - fastcall 133, 134
 - getch 142
 - interrupt 122
 - interrupt qualifier 122
 - jump table 133
 - kbhit 142
 - main 136, 169
 - nested 133

- putch 142
- recursion 105
- return values 131
- returning from 122
- root 169
- signatures 128, 174
- written in assembler 125

G

- getch function 142
- global directive 162
- global optimization 25, 104
- global psect flag 33, 165
- global symbols 180
- go to line 76
- grepping files 86

H

- hardware
 - initialization 137
 - stack 133
- header files 18
 - pic.h 113
 - problems in 103
- help
 - button 65
- help menu 88
 - C library reference 88
 - editor keys 89
 - HI-TECH Software 88
 - HTLPIC 88
 - PICL compiler options 90
 - release notes 90
 - technical support 89
- hex files 129
 - multiple 183

-
- hide
 - block 71, 72
 - button 65
 - HLINK
 - modifying options 44
 - Pspec 35
 - HLINK options 181
 - Aclass=low-high 38, 182
 - Cpsect=class 183
 - Dsymfile 183
 - Eerrfile 183
 - F 183
 - Gspec 184
 - H+symfile 184
 - Hsymfile 184
 - Jerrcount 184
 - K 185
 - L 185
 - LM 185
 - Mmapfile 185
 - N 185
 - Ns 185
 - Ooutfile 185
 - Pspec 185
 - Qprocessor 187
 - Sclass=limit[,bound] 187
 - Usymbol 187
 - Vavmap 188
 - Wnum 188
 - X 188
 - Z 188
 - HLINK options -Nc 185
 - hot keys 56
 - HTLPIC, table of 57
 - windows, table of 58
 - HTC_ERR_FORMAT 96
 - HTC_WARN_FORMAT 96
 - HTLPIC 51
 - <<>> menu 73
 - colours 53
 - attributes, table of 54
 - settings, table of 55
 - values, table of 54
 - command line arguments 52
 - editor 65
 - external ROM addresses 78
 - hardware requirements 52
 - help 88
 - hot keys 56
 - hot keys, table of 57
 - initialization file 53, 59
 - licence details 73
 - loading project file 52
 - menu bar 53
 - menus 73
 - mouse driver 53
 - moving windows 58
 - projects 81
 - pull down menus 53
 - quitting 74
 - resizing windows 58
 - screen mode 53
 - screen resolution 52
 - selecting windows 56
 - starting 51
 - tutorial 60
 - version number 59
 - window hot keys 58
 - windows 51, 52
 - htlpic.ini 53, 59

I

I/O

- console I/O functions 142
- serial 142
- STDIO 142

ICD support 78

ID locations 108, 135

idata_n psect 134

identifiers

- assembler 158
- length 80

idloc psect 135

IEEE floating point format 77, 114

if directive 169

Implementation-defined behaviour 121

- division and modulus 122
- shifts 121

include assembler control 175

indent

- block 71
- mode 67

indenting code 76

ini file 17, 53, 59, 129

- setting colours in 53

init psect 135

in-line assembly 123, 126

insert mode 66

instructions, bit 108

int data types 113

- accessing bits within 108

int_ret psect 123, 135

intcode psect 123, 135

Intel hex 129

intentry psect 123, 135

interrupt functions 122

- calling from main line code 124
- calling functions from 123

context retrieval 123

context saving 123, 141

midrange 122

returning from 122

interrupt keyword 122

interrupt level 124

interrupt_level directive 124

interrupts 122

handling in C 122

intsave psect 123, 136

intsave_n psect 123, 136

irp directive 172

irpc directive 172

J

Japanese character handling 138

JIS character handling 138

jis pragma directive 138

jmp_tab psect 135

jump tables 133, 135

K

kbhit function 142

keyword

auto 133

bank1 119

bank2 119

bank3 119

bit 112

const 117

disabling non-ANSI 103

extern 125

interrupt 122

persistent 118, 119, 120

volatile 117

L

- label field 160
- labels 158
 - ASPIC 158
 - local 171
 - re-defining 166
 - relocatable 162
- length of identifiers 80
- LIBR 192
 - command line arguments 192
 - error messages 195
 - listing format 194
 - long command lines 194
 - module order 194
- librarian 192
 - command files 194
 - command line arguments 192, 194
 - error messages 195
 - listing format 194
 - long command lines 194
 - module order 194
- libraries
 - adding files to 193
 - C reference 88
 - creating 193
 - default 92
 - deleting files from 193
 - format of 192
 - linking 187
 - listing modules in 193
 - module order 194
 - naming convention 105
 - order of 84
 - scanning additional 99
 - standard 105
 - standard, table of 106
 - used in executable 185
- library
 - difference between object file 192
 - manager 192
 - on-line manual 88
- licence
 - agreement 88
 - details 73
- line numbers
 - assembler 155
 - C source 81
- link addresses 32, 180, 185
- linker 28, 29, 179
 - command files 188
 - command line arguments 188
 - invoking 188
 - long command lines 188
 - modifying options 44
 - options from PICL 99
 - passes 192
 - symbols handled 180
- linker defined symbols 138
- linker errors
 - aborting 184
 - undefined symbols 185
- linker options 34, 85, 181
 - Aclass=low-high 38, 182, 186
 - Cpsect=class 183
 - Dsymfile 183
 - Eerrfile 183
 - F 183
 - Gspec 184
 - H+symfile 184
 - Hsymfile 184
 - I 185
 - Jerrcount 184
 - K 185
 - L 185
 - LM 185

- Mmapfile 185
- N 185
- Nc 185
- Ns 185
- numbers in 182
- Ooutfile 185
- P 35
- Pspec 185
- Qprocessor 187
- Sclass=limit[, bound] 187
- Usymbol 187
- Vavmap 188
- Wnum 188
- X 188
- Z 188
- linking programs 83, 129
- list files
 - assembler 26, 94
 - C source 21
 - generating 155
- list, assembler control 175
- little endian format 110, 113, 114
- ljmp mnemonic 155, 156, 174
- load addresses 32, 180, 185
- local directive 158, 171
- local psect flag 33, 165
- local psects 180
- local symbols 104, 173
 - suppressing 155, 188
- local variables 133
 - area size 169
 - auto 133
 - debugging information for 78
 - static 134
- location counter 158, 166
- long data types 113

M

- macro
 - calibration constant 110
 - configuration 107
 - ID location 108
- macro directive 160, 170
- macro names for DOS commands 88
- macros 170
 - ! character 174
 - % character 174
 - < and > characters 174
 - @ symbol 171
 - bitclr 109
 - bitset 109
 - concatenation of arguments 171
 - disabling in listing 176
 - expanding in listings 155, 175
 - invoking 174
 - nul operator 171
 - predefined 107
 - preprocessor 95
 - repeat with argument 172
 - suppressing comments 171
 - undefining 103
 - unnamed 172
- main function 136, 169
- make 82
- make menu 81
 - CPP include paths 85
 - CPP pre-defined symbols 84
 - library file list 84
 - linker options 85
 - load project 83
 - make 82
 - map file name 83
 - new project 83
 - object file list 84

- objtohex options 85
 - output file name 83
 - re-link 83
 - re-make 83
 - rename project 83
 - save project 83
 - source file list 84
 - symbol file name 84
 - mantissa 114
 - map file options 78
 - map files 102, 185
 - call graphs 190
 - example of 189
 - generating 100
 - naming 83
 - processor selection 187
 - segments 189
 - sorting symbols 78
 - symbol tables in 185
 - width of 188
 - memory
 - DOS 73
 - external ROM 78
 - specifying ranges 182
 - unused 185
 - usage 86, 102
 - menu
 - <<>> 59
 - compile 63, 78
 - edit 74
 - file 73
 - help 88
 - make 81
 - mouse operation 55
 - options 76
 - run 85
 - setup 53, 59
 - system 73
 - utility 85
 - menu bar 53
 - menus
 - accessing with keyboard 54
 - hot keys, for 56
 - HTLPIC 73
 - pull down 53
 - midrange pointers 119
 - mnemonics 156
 - mnemonics additional 156
 - modules
 - in library 192
 - list format 194
 - order in library 194
 - used in executable 185
 - mouse
 - auto-repeat rate 73
 - ballistic threshold 73
 - driver 53, 73
 - sensitivity 59
 - move block 70
 - moving windows 58
 - MPLAB
 - debugging information 78, 142
 - ICD support 78
 - symbol files 84
 - multiple hex files 183
 - multiple source files 81, 84
- ## N
- new files 73
 - next button 67
 - nocond assembler control 176
 - noexpand assembler control 176
 - nojis pragma directive 138
 - nolist assembler control 176
 - non-volatile RAM 118
 - noxref assembler control 176

numbers

- assembler 156
- in C source 111
- in linker options 182

numeric constants 156

numeric labels 159

nvrn 118

nvrn psect 135

O

object code, version number 185

object files 26, 84, 94

- absolute 26, 185
- displaying 26
- including line numbers 155
- precompiled 84
- relocatable 26, 179
- specifying object filenames 155
- suppressing local symbols 155
- symbol only 183

OBJTOHEX 28, 195

- command line arguments 195
- table of options 196

objtohex options 85

optimization 80

- assembler 25, 153, 155
- explanation of 88
- global 104
- peephole 25
- post-pass 100

option instruction 109

options menu 76

- fake local symbols 78
- floating point type 77
- long formats in printf 78
- map and symbol file options 78
- output file type 77
- ROM addresses 78

save dependency information 78

select processor 77

sort map by address 78

source level debug info 78

suppress local symbols 78

org directive 166

OSCCAL register 110

oscillator calibration constant 110

outdent block 71

outdenting code 76

output file formats 28, 185

HTLPIC 77

specifying 100, 195

table of 28, 107

overlaid psects 165

overlayed memory areas 185

overwrite mode 66

ovlrd psect flag 33

ovrld psect flag 165

P

p1 application 22

page

length 154

width 155

page assembler control 176

pages

chipinfo file 129

pagesel directive 174

parameter passing 125, 130, 131

parser 22

output 22

paste 72

peephole optimization 25

persistent keyword 118, 119, 120

persistent variables 135

PIC assembler language 156

functions 125

- pic.h 113
- picinfo.ini file 129, 154
- PICL
 - batch files 92
 - command format 91
 - file types 91
 - long command lines 91
 - options 92
 - predefined macros 107
 - redirecting options to 92
 - supported data types 110
- PICL options
 - AAHEX 92
 - ASMLIST 94
 - BIN 94, 129
 - C 94, 129
 - CR 95
 - D 95
 - D24 95, 114
 - D32 114
 - E 96, 97
 - Efile 97
 - FAKELOCAL 98, 143
 - FDOUBLE 98
 - G 98, 106
 - I 99
 - INTEL 99
 - L 99, 141
 - M 100
 - MOT 100
 - O 100, 106, 129
 - P 101
 - PRE 101
 - processor 92
 - PROTO 101
 - PSECTMAP 102, 129
 - q 103
 - S 103, 128, 129
 - SIGNED_CHAR 103, 113
 - STRICT 103
 - TEK 103
 - U 103
 - UBROF 104
 - V 104
 - W 104
 - X 104
 - Zg 104
- PICL output formats
 - American Automation Hex 92, 106
 - Binary 94, 106
 - Bytecraft 106
 - Intel Hex 99, 106
 - Motorola Hex 100, 106
 - Tektronix Hex 103, 106
 - UBROF 104, 106
- pointers
 - midrange 119
 - bank2 120
 - bank3 120
 - const 120
 - function 120
 - RAM 120
 - to const 121
- post-pass optimizer 100
- powerup psect 134
- powerup routine 92, 137
 - source for 137
- powerup symbol 136
- pragma directives 138
 - table of 140
- pre-compiled object files 84
- predefined symbols
 - preprocessor 107
- preprocessing 21, 79, 81, 101
 - assembler files 101

preprocessor

- macros 95
- output 21
- path 85, 99

preprocessor directives 138

- #asm 127
- #endasm 127
- table of 139

preprocessor symbols 84

- predefined 107

previous button 67

printf

- format checking 138
- long support 78

printf_check pragma directive 138

printing

- longs 78

processor selection 77, 92, 174, 187

program sections 160

project 83

project files 83, 88, 141

projects 81

- building 82
- creating 83
- defining preprocessor symbols 84
- libraries contained in 84
- linker options 85
- loading 83
- map file name 83
- object files contained in 84
- options in 76
- output file name 83
- path to include files 85
- re-building 83
- renaming 83
- saving 83
- source files contained in 84
- symbol file name 84

psect

- bss 180
- clrtext 135
- config 135
- constn 135
- cstrings 135
- ctextn 134
- data 180, 195
- end_init 135
- float_text 135
- idata_n 134
- idloc 135
- init 135
- int_ret 123, 135
- intcode 123, 135
- intentry 123, 135
- intsave 123, 136
- intsave_n 123, 136
- jmp_tab 135
- nvrn 135
- powerup 134
- rbit_n 136
- rbss_n 135
- rdata_n 135
- strings 135
- stringtable 135
- struct 136
- temp 136
- text 135
- textn 134
- xtemp 136

psect directive 30, 160, 164

psect flags 164, 187

psect pragma directive 44, 140

psects 25, 29, 160, 179

- absolute 162, 164, 165
- alignment 165
- basic kinds 179

- class 38, 182, 183, 187
- compiler generated 134
- default 162
- delta value of 45, 183
- differentiating ROM and RAM 165
- grouping 33
- linking 33, 179
- local 180
- maximum size of 165
- overlaid 33
- page placement 165
- positioning 33
- relocation 28
- renaming 140
- specifying address ranges 38, 186
- specifying addresses 35, 182, 185
- struct 132
- types of 31
- user defined 40, 140
- pseudo-function, float24 157
- pseudo-ops 162
 - table of 163
- pull down menus 53
- pure psect flag 165
- putch function 142

Q

- qualifiers
 - and auto variables 133
 - auto 133
 - bank1 118, 119
 - bank2 118, 119
 - bank3 118, 119
 - const 117, 120
 - fastcall 133
 - persistent 119, 120
 - volatile 117, 120
- quiet mode 103

- quitting HTLPIC 74

R

- radix specifiers
 - assembler 156
 - C source 111
- RAM
 - Bank 1 119
 - Bank 2 119
 - Bank 3 119
- RAM pointers 120
- rbit_n psect 136
- rbss_n psect 134, 135
- rdata_n psect 135
- read block 70
- recently opened files 74
- recursion 105
- redirecting errors 97
- redirecting options to PICL 92
- register
 - names 158
 - OSCCAL 110
 - SFR 158
 - usage 130
 - W 158
- regsused pragma directive 141
- release notes 90
- RELOC 184, 185
- reloc psect flag 165
- reloc psect flags 34
- relocatable
 - labels 162
 - object files 179
- relocatable listing file 27
- relocation 28, 179
- relocation information
 - preserving 185
- renaming psects 140

- replacing text 75
- rept directive 172
- reset
 - calibration constants 110
 - code executed after 137
- resizing windows 58
- RETFIE instruction 122
- RETLW instruction 122
- RETURN instruction 122
- return values 131
- ROM
 - access of objects in 118
 - pages 174
 - placing strings in 117
- ROM pages 155
- root functions 169
- run menu 85
 - DOS command 85
 - DOS shell 85
- runtime files 84
- runtime module 92, 136
 - source for 137

S

- saving files 74
- scroll bar 59
- search button 67
- search path
 - header files 99
- searching files 75, 86
- segment selector 184
- segments 43, 184, 189
- selecting text 71
- serial I/O 142
- set directive 160, 166
- setting bits 108
- setting tab size 76
- setup menu 53, 59
- SFRs 158
- shift operations
 - result of 121
- show clipboard 72, 76
- signat directive 128, 174
- signature checking 128
- signatures 174
- signed char variables 103
- size error message, suppressing 155
- size psect flag 165
- sound, enabling 59
- source files 84
- source listings 81
- source modules 21
- space assembler control 176
- space psect flag 165
- special characters 157
- special function registers 158
- sprintf
 - long support 78
- S-Record files 100
- stack, hardware 133
- standard libraries 105
 - table of 106
- standard symbols 85
- start symbol 136
- startup module 84, 92, 136
 - clearing bss 180
 - data copying 180
- statements
 - assembler 160
- static variables 134
- status line
 - indent/C mode indicator 67
 - insert/overwrite indicator 66
 - WordStar indicator 66
- STDIO 142
- string search 75, 86

- strings 117
 - assembler 160
 - placement 117, 118
- strings psect 135
- stringtable psect 135
- struct psect 132, 136
- structures 115
 - bit fields 116
 - qualifiers 115
- subtile assembler control 176
- symbol
 - power up 136
 - start 136
- symbol files 98, 106
 - debug info 78
 - enhanced 184
 - generating 184
 - local symbols in 188
 - MPLAB specific 98
 - naming 84
 - old style 183
 - options 78
 - producing MPLAB specific 78
 - removing local symbols from 78, 104
 - removing symbols from 187
 - source level 98
- symbol tables 106, 185, 187
 - sorting 185
 - sorting addresses 78
- symbolic labels 158
- symbols
 - ASPIC generated 158
 - assembler 158
 - equating 166
 - global 180, 193
 - linker defined 138
 - MPLAB specific 143
 - pre-defined 84
 - undefined 187

- syntax highlighting 76
- system menu 73

T

- tab size 76
- technical support 89
- Tektronix hex files 103
- temp path 18, 52
- temp psect 136
- text psect 135
- text search 75, 86
- textn psect 134
- title assembler control 176
- TRIS instruction 109
- TSR programs 85
- tutorial
 - compiling 63
 - errors 64
 - getting started 60
- typographic conventions 15

U

- UBROF files 104
- uncomment block 71
- uncommenting code 76
- undefined symbols
 - assembler 155
- unions 115
- utilities 179
- utility menu 85
 - ascii table 87
 - calculator 86
 - define user commands 87
 - memory usage map 86
 - string search 86

V

variable argument list 130

variables

- absolute 43, 115

- accessing from assembler 126

- auto 133

- bit 112

- char types 113

- floating point types 114

- int types 113

- local 133

- persistent 135

- static 134

verbose 104

version number 59

video card information 73

volatile keyword 117, 120

with psect flag 165

word boundaries 165

WordStar

- block commands 69

- indicator 66

write block 71

X

xref assembler control 177

xtemp psect 136

Z

zoom 59

zoom command 67

W

W register 130, 158

warning level 80, 104

- setting 188

warnings 79

- level displayed 104

- suppressing 188

window

- edit 66

- error 64

windows

- buttons in 59

- moving 58

- resize/move hot key 58

- resizing 58

- scroll bar in 59

- selecting 56

- zooming 59

HPDPIC menu hot keys

Key	Meaning
Alt-O	Open editor file
Alt-N	Clear editor file
Alt-S	Save editor file
Alt-A	Save editor file with new name
Alt-Q	Quit to DOS
Alt-J	DOS Shell
Alt-F	Open File menu
Alt-E	Open Edit menu
Alt-I	Open Compile menu
Alt-M	Open Make menu
Alt-R	Open Run menu
Alt-T	Open Options menu
Alt-U	Open Utility menu
Alt-H	Open Help menu
Alt-P	Open Project file
Alt-W	Warning level dialog
Alt-Z	Optimization menu
Alt-D	Command.com
F3	Compile and link single file
Shift-F3	Compile to object file
Ctrl-F3	Compile to assembler code
Ctrl-F4	Retrieve last file
F5	Make target program
Shift-F5	Re-link target program
Ctrl-F5	Re-make all objects and target program
Alt-P	Load project file
Shift-F7	User defined command 1
Shift-F8	User defined command 2
Shift-F9	User defined command 3
Shift-F10	User defined command 4
F2	Search in edit window
Alt-X	Cut to clipboard
Alt-C	Copy to clipboard
Alt-V	Paste from clipboard

PICC Options

Option	Meaning
<i>-processor</i>	Define the processor
<i>-A-option</i>	Specify <i>-option</i> to be passed directly to the assembler
-AAHEX	Generate an American Automation symbolic HEX file
-ASMLIST	Generate assembler .LST file for each compilation
-BIN	Generate a Binary output file
-C	Compile to object files only
-CKfile	Make OBJTOHEX use a checksum file
-CRfile	Generate cross-reference listing
-D24	Use truncated 24-bit floating point format for doubles
-D32	Use IEEE754 32-bit floating point format for doubles
-Dmacro	Define pre-processor macro
-E	Use “editor” format for compiler errors
-Efile	Redirect compiler errors to a file
-E+file	Append errors to a file
-FAKELOCAL	Produce MPLAB-specific debug information
-FDOUBLE	Enables the use of faster 32-bit floating point math routines.
-Gfile	Generate enhanced source level symbol table
-HELP	Print summary of options
-Ipath	Specify a directory pathname for include files
-INTEL	Generate an Intel HEX format output file (default)
-Llibrary	Specify a library to be scanned by the linker
-L-option	Specify <i>-option</i> to be passed directly to the linker
-Mfile	Request generation of a MAP file
-MOT	Generate a Motorola S1/S9 HEX format output file
-Nsize	Specify identifier length
-NORT	Do not link standard runtime module
-O	Enable post-pass optimization
-Ofile	Specify output filename
-P	Preprocess assembler files
-PRE	Produce preprocessed source files
-PROTO	Generate function prototype information
-PSECTMAP	Display complete memory segment usage after linking
-q	Specify quiet mode
-RESRAM=ranges	Reserve the specified RAM address ranges.
-RESROM=ranges	Reserve the specified ROM address ranges.
-ROMranges	Specify external ROM memory range available
-S	Compile to assembler source files only
-SIGNED_CHAR	Make the default char signed.
-STRICT	Enable strict ANSI keyword conformance

PICC Options

Option	Meaning
-TEK	Generate a Tektronix HEX format output file
-U <i>symbol</i>	Undefine a predefined pre-processor symbol
-UBROF	Generate an UBROF format output file
-V	Verbose: display compiler pass command lines
-W <i>level</i>	Set compiler warning level
-X	Eliminate local symbols from symbol table
-Zg	Enable global optimization in the code generator