user's guide

user's guide

XA C Manual

Sixth Printing, November 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 HPDXA 3

XAC Command Line Compiler Driver 4

Features and Run-time Environment 5

The XA Macro Assembler 6

Linker and Utilities Reference Manual 7

Lucifer source level debugger 8

Error messages 9

Library functions 10

1 - Introduction - 17

1.1 Typographic Conventions - 17
1.2 Using This Manual - 17

2 - Tutorials - 19

2.1 Overview of the compilation process - 19
2.1.1 Compilation - 19
2.1.2 The compiler input - 20
2.1.2.1 Steps before linking - 23
2.1.2.2 The link stage - 29
2.2 Psects and the linker - 31
2.2.1 Psects - 31
2.2.1.1 The psect directive - 32
2.2.1.2 Psect types - 33
2.3 Linking the psects - 35
2.3.1 Grouping psects - 35
2.3.2 Positioning psects - 35
2.3.3 Linker options to position psects - 36
2.3.3.1 Placing psects at an address - 37
2.3.3.2 Exceptional cases - 40
2.3.3.3 Psect classes - 40
2.3.3.4 User-defined psects - 42
2.3.4 Issues when linking - 43
2.3.4.1 Paged memory - 44
2.3.4.2 Separate memory areas - 45
2.3.4.3 Objects at absolute addresses - 45
2.3.5 Modifying the linker options - 46

3 - Using HPDXA - 49

3.1 Introduction - 49
3.1.1 Starting HPDXA - 49
3.2 The HI-TECH Windows User Interface - 50
3.2.1 Environment variables - 50
3.2.2 Hardware Requirements - 50

3.2.3 Colours - 51
3.2.4 Pull-down menus - 51
3.2.4.1 Keyboard menu selection - 52
3.2.4.2 Mouse menu selection - 53
3.2.4.3 Menu hot keys - 54
3.2.5 Selecting Windows - 54
3.2.6 Moving and Resizing Windows - 56
3.2.7 Buttons - 57
3.2.8 The Setup menu - 57
3.3 Tutorial: Creating and compiling a C program - 58
3.4 The HPDXA Editor - 63
3.4.1 Frame - 64
3.4.2 Content region - 64
3.4.3 Status line - 64
3.4.4 Keyboard Commands - 65
3.4.5 Block commands - 65
3.4.6 Clipboard editing - 69
3.4.6.1 Selecting Text - 69
3.4.6.2 Clipboard commands - 69
3.5 HPDXA menus - 70
3.5.1 <<>> menu - 70
3.5.2 File menu - 71
3.5.3 Edit menu - 72
3.5.4 Options menu - 74
3.5.5 Compile menu - 78
3.5.6 Make menu - 80
3.5.7 Run menu - 84
3.5.8 Utility menu - 86
3.5.9 Help menu - 88

4 - XAC Command Line Compiler Driver - 91

4.1 Long Command Lines - 91
4.2 Default Libraries - 92
4.3 Standard Run-Time Code - 92
4.4 XAC Compiler Options - 92
4.4.1 -Aspec: Set ROM and RAM Addresses - 92
4.4.2 -AAHEX: Generate American Automation Symbolic HEX - 96

4.4.3 -ASMLIST: Generate Assembler .LST Files - - - - - - - - - - - - - - - - - 96
4.4.4 -AV: Select Avocet Symbol File - 96
4.4.5 -BIN: Generate Binary Output File - 96
4.4.6 -Bh: Select Huge Memory Model - 96
4.4.7 -Bl: Select Large Memory Model - 96
4.4.8 -Bm: Select Medium Memory Model - 98
4.4.9 -Bs: Select Small Memory Model - 98
4.4.10 -C: Compile to Object File - 98
4.4.11 -CRfile: Generate Cross Reference Listing - - - - - - - - - - - - - - - - 99
4.4.12 -CLIST: Produce C Listing File - 99
4.4.13 -Dmacro: Define Macro -100
4.4.14 -DOUBLE -100
4.4.15 -E*[[+]file]*: Define Format for Compiler Errors / Redirect Errors - - - - - -100
4.4.15.1 Using the -E Option -101
4.4.15.2 Modifying the Standard -E Format - - - - - - - - - - - - - - -101
4.4.15.3 Redirecting Errors to a File - - - - - - - - - - - - - - - - - -102
4.4.16 -FDOUBLE: Select Fast Double Precision - - - - - - - - - - - - - - - -103
4.4.17 -Gfile: Generate source-level Symbol File - - - - - - - - - - - - - - -103
4.4.18 -Hfile: Generate Assembler Level Symbol File - - - - - - - - - - - - - -103
4.4.19 -HELP: Display Help -103
4.4.20 -INTEL: Generate INTEL Hex File -103
4.4.21 -Ipath: Include Search Path -103
4.4.22 -Llibrary: Scan Library -104
4.4.23 -L-option: Specify Extra Linker Option - - - - - - - - - - - - - - - - -105
4.4.24 -Mfile: Generate Map File -105
4.4.25 -MOTOROLA: Generate Motorola S-Record HEX File - - - - - - - - - - -105
4.4.26 -Nlength: Specify Identifier Significant Length - - - - - - - - - - - - -105
4.4.27 -O: Invoke Optimizer -105
4.4.28 -Ofile: Specify Output File -106
4.4.29 -OMF51: Produce OMF-51 Output File - - - - - - - - - - - - - - - - - -106
4.4.30 -P: Preprocess Assembly Files -106
4.4.31 -PRE: Produce preprocessed Source Code - - - - - - - - - - - - - - - -106
4.4.32 -PROTO: Generate Prototypes -106
4.4.33 -PSECTMAP: Display Complete Memory Usage - - - - - - - - - - - - - -107
4.4.34 -q: Quiet Mode -108
4.4.35 -RAMranges: Define RAM Ranges -108
4.4.36 -ROMranges: Define ROM Ranges -108
4.4.37 -S: Compile to Assembler Code -109

4.4.38 -SA: Compile to Avocet assembler source files - - - - - - - - - - - - - - - 109
4.4.39 -STRICT: Strict ANSI Conformance - 109
4.4.40 -TEK: Generate Tektronix HEX File - 109
4.4.41 -Umacro: Undefine a Macro - 109
4.4.42 -UBROF: Generate UBROF Format Output File - - - - - - - - - - - - - - 109
4.4.43 -UNSIGNED: Make *char* Type Unsigned - - - - - - - - - - - - - - - - - 110
4.4.44 -V: Verbose Compile - 110
4.4.45 -Wlevel: Set Warning Level - 110
4.4.46 -X: Strip Local Symbols - 110
4.4.47 -Zg*[level]*: Global Optimisation - 110

5 - Features and Run-time Environment - - - - - - - - - - - - - - 111

5.1 ANSI Standard Issues - 111
5.1.1 Implementation-defined behaviour - 111
5.2 Processor-related Features - 111
5.2.1 Code Size Limitations of Functions and Modules - - - - - - - - - - - - - 111
5.2.2 XA Hardware Design - 111
5.3 Files - 112
5.3.1 Source Files - 112
5.3.2 Output File Formats - 112
5.3.3 Symbol Files - 112
5.3.4 Avocet Symbol Tables - 113
5.3.5 Standard Libraries - 113
5.3.6 Run-time Startup Modules - 114
5.3.6.1 Stack pointer and XA memory model - - - - - - - - - - - - - - - - 114
5.3.6.2 Clearing the bss, rbss and rbit psects - - - - - - - - - - - - - - - 115
5.3.6.3 Memory models and the data Psect - - - - - - - - - - - - - - - - - 116
5.3.7 Customising the Run-time Startup Code - - - - - - - - - - - - - - - - - 116
5.3.7.1 Copyright Notice - 116
5.3.8 Using the New Run-time Module Code - - - - - - - - - - - - - - - - - - 116
5.3.9 The powerup Routine - 117
5.4 Supported Data Types and Variables - 117
5.4.1 Radix Specifiers and Constants - 118
5.4.2 Bit Data Types - 119
5.4.2.1 Using Bit Addressable SFRs - 120
5.4.3 8-Bit Integer Data Types - 121
5.4.4 16-Bit Integer Data Types - 121

5.4.5 32-Bit Integer Data Types -122
5.4.6 Floating-point -122
5.4.7 Structures and Unions -123
5.4.8 Bit Fields in Structures -123
5.4.8.1 Structure and Union Qualifiers -124
5.4.9 Standard Type Qualifiers -125
5.4.9.1 Const and Volatile Type Qualifiers - - - - - - - - - - - - - - - - - - -125
5.4.10 Special Type Qualifiers -125
5.4.10.1 Near Qualifier -126
5.4.10.2 Far Qualifier -126
5.4.10.3 Persistent Qualifier -126
5.4.11 Code Qualifier -127
5.4.12 Pointer Types -128
5.4.12.1 Combining Type Qualifiers and Pointers - - - - - - - - - - - - - - -129
5.4.13 Code Pointers -130
5.4.14 Const Pointers -130
5.4.14.1 Huge Pointer Qualifier -131
5.5 Storage Class and Object placement -131
5.5.1 Local Variables -131
5.5.1.1 Auto Variables -131
5.5.1.2 Static Variables -132
5.5.2 Absolute Variables -132
5.6 Functions -133
5.6.1 The noregsave Function Qualifier -133
5.6.2 Function Argument Passing -133
5.6.3 Function Return Values -133
5.6.3.1 8-Bit Return Values -133
5.6.3.2 16-Bit Return Values -134
5.6.3.3 32-Bit Return Values -134
5.6.3.4 64-Bit Return Values -134
5.6.3.5 Structure Return Values -134
5.7 Memory Usage -135
5.8 Register Usage -135
5.9 Operators -135
5.9.1 Integral Promotion -135
5.9.2 Shifts applied to integral types -137
5.9.3 Division and modulus with integral types - - - - - - - - - - - - - - - - -137
5.10 Psects -137

5.10.1 Compiler-Generated Psects - 138
5.11 Interrupt Handling in C - 139
5.11.1 Interrupt Functions - 140
5.11.2 Interrupt Priorities - 140
5.11.3 Banked Interrupts - 141
5.11.4 Interrupt Handling Macros - 141
5.11.5 Enabling and Disabling Interrupts - 141
5.11.6 ROM_VECTOR - 142
5.11.7 RAM-Based Interrupt Vectors - 142
5.11.7.1 RAM_VECTOR - 144
5.11.7.2 CHANGE_VECTOR - 144
5.11.7.3 READ_RAM_VECTOR - 144
5.11.8 Trap Interrupts - 145
5.11.9 Predefined Interrupt Vector Names - - - - - - - - - - - - - - - - - - - 147
5.12 Mixing C and XA Assembler Code - 147
5.12.1 External Assembly Language Functions - - - - - - - - - - - - - - - - - 147
5.12.2 Accessing C objects from within assembler - - - - - - - - - - - - - - - 148
5.12.3 #asm, #endasm and asm() - 149
5.13 Preprocessing - 150
5.13.1 Preprocessor Directives - 150
5.13.2 Predefined Macros - 150
5.13.3 Pragma Directives - 150
5.13.3.1 The #pragma jis and nojis Directives - - - - - - - - - - - - - - - 150
5.13.3.2 The #pragma pack Directive - - - - - - - - - - - - - - - - - - - 150
5.13.3.3 The #pragma printf_check Directive - - - - - - - - - - - - - - - 152
5.13.3.4 The #pragma psect Directive - - - - - - - - - - - - - - - - - - 152
5.13.3.5 Loading Code in Large Model at Specific Addresses - - - - - - - 155
5.13.4 The #pragma strings Directive - 155
5.13.5 The #pragma switch Directive - 155
5.14 Linking Programs - 156
5.14.1 Replacing Library Modules - 156
5.14.2 Signature Checking - 157
5.14.3 Linker-Defined Symbols - 158
5.15 Optimising Code for the XA - 158
5.16 Standard I/O Functions and Serial I/O - 158

6 - The XA Macro Assembler -161

6.1 Assembler Usage -161
6.2 Assembler Options -161
6.3 XA Assembly Language -163
6.3.1 Character set -163
6.3.2 Numeric Constants -163
6.3.3 Delimiters -164
6.3.4 Special Characters -164
6.3.5 Identifiers -164
6.3.5.1 Significance of Identifiers -164
6.3.5.2 SFR Names -165
6.3.5.3 Assembler Generated Identifiers -165
6.3.5.4 Location Counter -165
6.3.5.5 Register Symbols -165
6.3.6 Strings -165
6.3.7 Temporary Labels -165
6.3.8 Expressions -166
6.3.9 The Bit Operator -166
6.3.10 Statement Format -167
6.3.11 Addressing Modes -168
6.3.12 Program Sections -168
6.3.13 Assembler Directives -168
6.3.13.1 PUBLIC -168
6.3.13.2 EXTRN -168
6.3.13.3 GLOBAL -168
6.3.13.4 END -169
6.3.13.5 PSECT -169
6.3.13.6 ORG -171
6.3.13.7 EQU and SET -171
6.3.13.8 DB and DW -172
6.3.13.9 DF and DD -172
6.3.13.10 DS -172
6.3.13.11 IF, ELSE, ELSEIF and ENDIF -172
6.3.13.12 MACRO and ENDM -173
6.3.13.13 LOCAL -174
6.3.13.14 REPT -174
6.3.13.15 IRP and IRPC -175

6.3.13.16 .SIGNAT - 176
6.3.14 Macro Invocations - 176
6.3.15 Assembler Controls - 177
6.3.15.1 COND - 177
6.3.15.2 EJECT - 177
6.3.15.3 GEN - 177
6.3.15.4 INCLUDE(pathname) - 178
6.3.15.5 LIST - 178
6.3.15.6 NOCOND - 178
6.3.15.7 NOGEN - 178
6.3.15.8 NOLIST - 178
6.3.15.9 NOXREF - 178
6.3.15.10 PAGELENGTH(n) - 178
6.3.15.11 PAGEWIDTH(n) - 178
6.3.15.12 RESTORE - 178
6.3.15.13 SAVE - 178
6.3.15.14 TITLE(string) - 178
6.3.15.15 XREF - 179

7 - Linker and Utilities Reference Manual - - - - - - - - - - - - - 181

7.1 Introduction - 181
7.2 Relocation and Psects - 181
7.3 Program Sections - 181
7.4 Local Psects - 182
7.5 Global Symbols - 182
7.6 Link and load addresses - 182
7.7 Operation - 183
7.7.1 Numbers in linker options - 184
7.7.2 -Aclass=low-high,... - 184
7.7.3 -Cx - 185
7.7.4 -Cpsect=class - 185
7.7.5 -Dclass=delta - 185
7.7.6 -Dsymfile - 185
7.7.7 -Eerrfile - 185
7.7.8 -F - 185
7.7.9 -Gspec - 186
7.7.10 -Hsymfile - 186

7.7.11 -H+symfile -186
7.7.12 -Jerrcount -186
7.7.13 -K -187
7.7.14 -I -187
7.7.15 -L -187
7.7.16 -LM -187
7.7.17 -Mmapfile -187
7.7.18 -N, -Ns and-Nc -187
7.7.19 -Ooutfile -187
7.7.20 -Pspec -187
7.7.21 -Qprocessor -189
7.7.22 -S -189
7.7.23 -Sclass=limit[, bound] -189
7.7.24 -Usymbol -189
7.7.25 -Vavmap -190
7.7.26 -Wnum -190
7.7.27 -X -190
7.7.28 -Z -190
7.8 Invoking the Linker -190
7.9 Map Files -191
7.9.1 Call Graph Information -192
7.10 Librarian -194
7.10.1 The Library Format -194
7.10.2 Using the Librarian -195
7.10.3 Examples -196
7.10.4 Supplying Arguments -196
7.10.5 Listing Format -196
7.10.6 Ordering of Libraries -197
7.10.7 Error Messages -197
7.11 Objtohex -197
7.11.1 Checksum Specifications -197
7.12 Cref -198
7.12.1 -Fprefix -199
7.12.2 -Hheading -199
7.12.3 -Llen -199
7.12.4 -Ooutfile -199
7.12.5 -Pwidth -200
7.12.6 -Sstoplist -200

7.12.7 -Xprefix - - - - - 200
7.13 Cromwell - - - - - 200
7.13.1 -Pname - - - - - 201
7.13.2 -D - - - - - 201
7.13.3 -C - - - - - 201
7.13.4 -F - - - - - 202
7.13.5 -Okey - - - - - 202
7.13.6 -Ikey - - - - - 202
7.13.7 -L - - - - - 202
7.13.8 -E - - - - - 202
7.13.9 -B - - - - - 202
7.13.10 -M - - - - - 202
7.13.11 -V - - - - - 202
7.14 Memmap - - - - - 202
7.14.1 Using MEMMAP - - - - - 203
7.14.1.1 -P - - - - - 203
7.14.1.2 -Wwid - - - - - 203

8 - Lucifer source level debugger - - - - - 205

8.1 Using Lucifer - - - - - 205
8.2 Symbol Names In Expressions - - - - - 206
8.2.1 Auto Variables and Parameters - - - - - 207
8.3 Lucifer Command Set - - - - - 207
8.3.1 The B command: set or display breakpoints - - - - - 207
8.3.2 The C command: display instruction at PC - - - - - 208
8.3.3 The D command: display memory contents - - - - - 209
8.3.4 The E command: examine C source code - - - - - 209
8.3.5 The G command: commence execution - - - - - 209
8.3.6 The I command: toggle instruction trace mode - - - - - 209
8.3.7 The L command: load a hex file - - - - - 210
8.3.8 The M command: modify memory - - - - - 210
8.3.9 The Q command: exit to operating system - - - - - 210
8.3.10 The R command: remove breakpoints - - - - - 211
8.3.11 The S command: step one line - - - - - 211
8.3.12 The T command: trace one instruction - - - - - 212
8.3.13 The U command: disassemble machine instructions - - - - - 212
8.3.14 The W command: upload binary - - - - - 212

8.3.15 The X command: examine or change registers - - - - - - - - - - - - - - -212
8.3.16 The @ command: display C variables - - - - - - - - - - - - - - - - - - -213
8.3.17 The . command: set a breakpoint and go - - - - - - - - - - - - - - - - - -213
8.3.18 The ; command: display from a source line - - - - - - - - - - - - - - - - -214
8.3.19 The = command: display next page of source - - - - - - - - - - - - - - - -215
8.3.20 The - command: display previous page of source - - - - - - - - - - - - - -215
8.3.21 The / command: search source file for a string - - - - - - - - - - - - - - -215
8.3.22 The ! command: execute a DOS command - - - - - - - - - - - - - - - - -215
8.3.23 Other commands -215
8.4 User Input and Output with Lucifer -216
8.5 Installing Lucifer on a Target -216
8.5.1 Modifying the Target Code -216
8.5.2 Memory Mapping -216
8.5.3 Interrupts -217

9 - Error messages -219

10 - Library functions -271

11 - Index -367

Table 2 - 1 - Configuration files 20
Table 2 - 2 - Input file types 22
Table 2 - 3 - clist output 23
Table 2 - 4 - preprocessor output. 24
Table 2 - 5 - Intermediate and Support files 25
Table 2 - 6 - Parser output 25
Table 2 - 7 - Code generator output. 26
Table 2 - 8 - Assembler output 28
Table 2 - 9 - Assembler listing 29
Table 2 - 10 - Output formats 31
Table 3 - 1 - Colour values 52
Table 3 - 2 - Colour attributes 52
Table 3 - 3 - Colour coding settings 53
Table 3 - 4 - Menu system key and mouse actions 53
Table 3 - 5 - HPDXA menu hot keys 55
Table 3 - 6 - Resize mode keys 56
Table 3 - 7 - Block operation keys 66
Table 3 - 8 - Editor keys 67
Table 3 - 9 - Macros usable in user commands 87
Table 4 - 1 - XAC File Types 91
Table 4 - 2 - XAC Options 93
Table 4 - 3 - Huge Model Libraries. 97
Table 4 - 4 - Large Model Libraries 97
Table 4 - 5 - Medium Model Libraries 98
Table 4 - 6 - Small Model Libraries 99
Table 4 - 7 - Error Format Specifiers 101
Table 5 - 1 - Output File Formats 113
Table 5 - 2 - Standard Run-time Startup Modules 116
Table 5 - 3 - Data Types 118
Table 5 - 4 - Radix Formats. 119
Table 5 - 5 - IEEE 32-bit Floating-point Format. 122
Table 5 - 6 - IEEE 64 bit Floating-point Format. 122
Table 5 - 7 - 64 bit Fast Double Format 123

Table 5 - 8 - XAC Pointer sizes (in bytes) and memory models 128
Table 5 - 9 - Integral division 137
Table 5 - 10 - Interrupt Support Macros and Functions 141
Table 5 - 11 - Interrupt PSW Values 143
Table 5 - 12 - Preprocessor directives 151
Table 5 - 13 - Interrupt vectors 153
Table 5 - 14 - Predefined CPP Symbols 154
Table 5 - 15 - Pragma Directives 154
Table 5 - 16 - Supported STDIO Functions. 159
Table 6 - 1 - ASXA Assembler options. 162
Table 6 - 2 - ASXA Numbers and bases 164
Table 6 - 3 - Operators. 167
Table 6 - 4 - ASXA Statement formats 167
Table 6 - 5 - ASXA Directives (pseudo-ops) 169
Table 6 - 6 - PSECT flags 170
Table 6 - 7 - : ASXA Assembler controls 177
Table 7 - 1 - Linker Options 183
Table 7 - 2 - Librarian Options 195
Table 7 - 3 - Librarian Key Letter Commands 195
Table 7 - 4 - Objtohex Options 198
Table 7 - 5 - Cref Options 199
Table 7 - 6 - Format Types 200
Table 7 - 7 - Cromwell Options. 201
Table 7 - 8 - Memmap options 203
Table 8 - 1 - Lucifer expression forms 207
Table 8 - 2 - Lucifer Command Set 208
Table 8 - 3 - Lucifer @ command variants 214

Figure 2 - 1 - Compilation overview 21
Figure 3 - 1 - HPDXA Startup Screen 49
Figure 3 - 2 - Setup Dialogue 58
Figure 3 - 3 - Hello program in HPDXA 59
Figure 3 - 4 - HPDXA File Menu 60
Figure 3 - 5 - ROM and RAM Address dialog 61
Figure 3 - 6 - Error window 62
Figure 3 - 7 - HPDXA Edit Menu 72
Figure 3 - 8 - Options Menu 75
Figure 3 - 9 - HPDXA Compile Menu 79
Figure 3 - 10 - HPDXA Make Menu 81
Figure 3 - 11 - HPDXA Run Menu 85
Figure 3 - 12 - HPDXA Utility Menu 86
Figure 3 - 13 - HPDXA Help Menu 88
Figure 4 - 1 - Library Prefixes and Suffixes 104
Figure 5 - 1 - XA Standard Library Naming Convention 114

Introduction

1.1 Typographic Conventions

Different fonts and styles are used throughout this manual to indicate special words or text. Computer prompts, responses and filenames will be printed in `constant-spaced type`. Samples of code, C keywords or types, assembler instruction and labels will be printed in a **`bold constant-space type`**. Particularly useful points and new terms will be emphasised using *italicised type*. With a window-based program like HPDXA, some concepts are difficult to convey in text. These will be introduced using short tutorials and sample screen displays with references to menu items, dialogue tab labels and button names in **bold text**.

1.2 Using This Manual

This manual is a comprehensive guide and reference to using XA 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 XA C command-line interface
- ❒ In-depth description of the C compiler
- ❒ How to use the assembler
- ❒ How to use the linker and other utilities
- ❒ Using Lucifer, the remote source-level debugger
- ❒ Error messages and their meaning
- ❒ Description of provided library functions

For information on installing XA C, using the on-line manual and getting started, see the *Quick Start Guide*.

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 20. 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 of your distribution. 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 stage and the files that are involved. Reference should be made to Figure 2 - 1 on page 21 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 extension[2] 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.

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

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

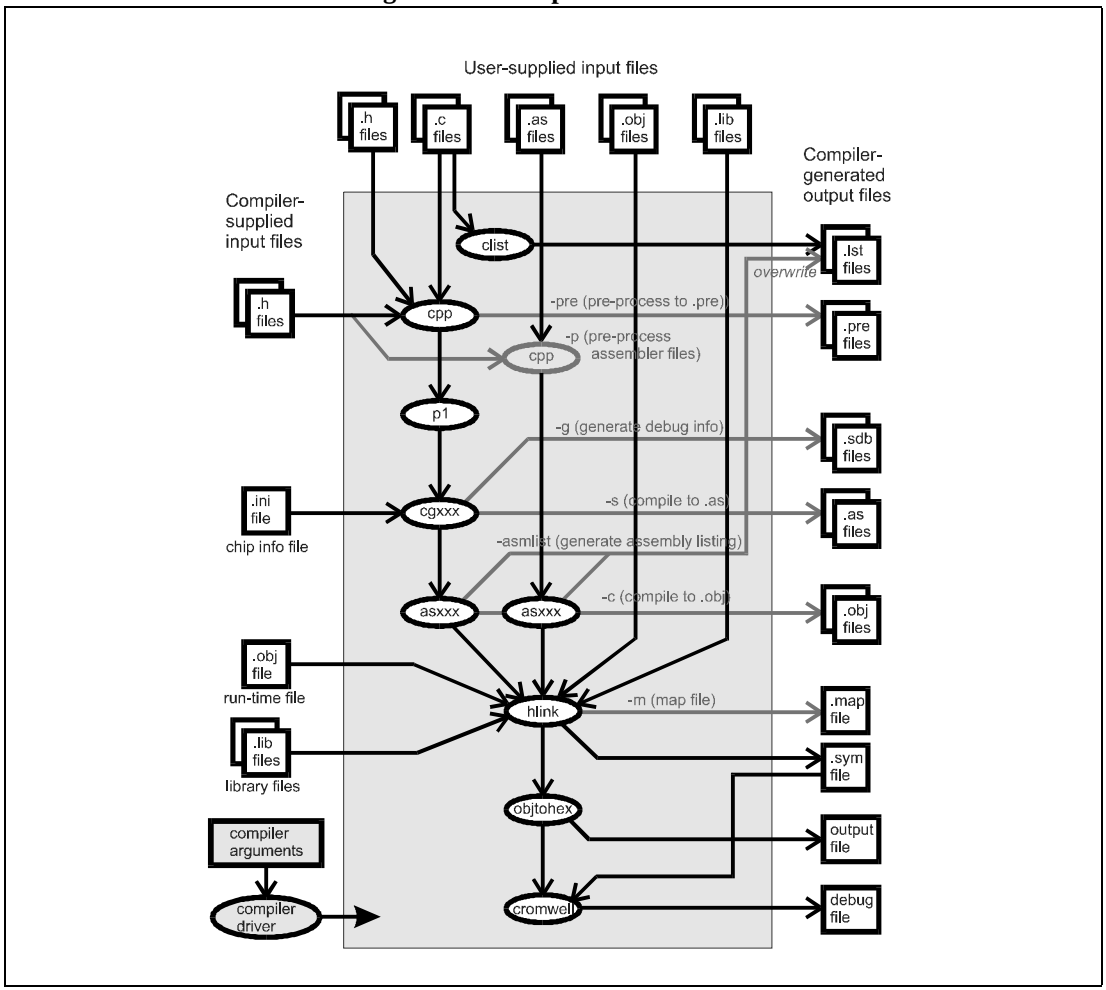
The user supplies several things to the compiler to make a program: the input files and the compiler options, whether using the CLD or HPD. The compiler accepts many different input file types. These are discussed below.

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 preprocessor directive and are often called *include files*. Since header files are referenced by a command that includes

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



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 23. 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. For example, if you have included `init.c` into the project, you should *not* include `init.obj` into the object file list.

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 the LIB 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 proceeded 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 23.

Table 2 - 3 clist output

C source	C listing

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

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 preprocessor directives in the source. Directives may, for example, replace macros with their replacement text (e.g. **`#define`** directives) or conditionally include source code subject to certain conditions (e.g. **`#if`**, **`#ifdef`** etc. directives). The preprocessor also inserts header files, whether user- or compiler-supplied, into the source. Table 2 - 4 on page 24 shows preprocessor output for the test program.

The output of the preprocessor is C source, but it may contain code which has been included by the preprocessor from header files and conditional code may have been omitted. Thus the preprocessor output usually contains similar, but different code to the original source file. The preprocessor 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 operates 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 preprocessor output for the test program is included by the preprocessor to indicate the filename 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 preprocessor is controlled by the compiler driver (either the CLD or

Table 2 - 4 preprocessor output

C source	Pre-processed output

`#define VAL 2`
 	
`int a, b = 1;`
 	
`void`
`main(void)`
`{`
`  /* set starting value */`
`  a = b + VAL;`
`}`
	`# 1 "main.c"`
	
	`int a, b = 1;`
	
	`void`
	`main(void)`
	`{`
	
	`a = b + 2;`
	`}`

HPD). The type of information that the driver supplies the preprocessor 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 preprocessor can evaluate preprocessor directives which contain a **sizeof(*type*)** expression. The output of the preprocessor 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 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 C 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 25 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

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

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.

Table 2 - 6 Parser output

C source	Parsed output

`#define VAL 2`
 	
`int a, b = 1;`
 	
`void`
`main(void)`
`{`
`    /* set starting value */`
`    a = b + VAL;`
`}`
	`Version 3.2 HI-TECH Softwa...`
	`"3 main.c`
	`[v _a 'i 1 e ]`
	`[v _b 'i 1 e ]`
	`[i _b`
	`-> 1 '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 ]`
	`}`

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

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

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 used to generate code for the statement highlighted in Table 2 - 7 on page 26 using 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 26 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 25 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 and has a processor-dependent name such as ASPIC 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

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.

Table 2 - 8 Assembler output

C source	Relocatable object file

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

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 28 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 29. 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 21 in the interests of clarity.

Table 2 - 9 Assembler listing

C source	Assembler listing

`#define VAL 2`
 	
`int a, b;`
 	
`void`
`main(void)`
`{`
`    /* set start...`
`    a = b + VAL;`
`}`
	`10  0000'                psect text`
	`11  0000'              _main:`
	`12                         ;main.c: 9: a = b + 2;`
	`13  0000' 99 08 0000'      mov.w   r0,#_b`
	`14  0004' 88 10            movc.w  r1,[r0+]`
	`15  0006' A9 12            adds.w  r1,#2`
	`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 unresolved addresses and address markers removed and replaced with their final 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 was 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 preprocessor allowing the using of all preprocessor directives within assembly code. The output of the preprocessor 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 relocatable 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]

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

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 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. For example, if the address of the symbol **_b** in our running example was determined to be 20000h, the linker would not be able to fit this address into the first underlined two byte "hole" in the object file shown dumped in the Table Assembler output on page 28 since 20000h is larger than two bytes long.

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.

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.

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 Automation hex	code and symbol information in binary, American Automation 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 Format 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 variables to be placed in non-volatile areas, 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 **p**rogram **sect**ions. 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 second **`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 can be linked in three different ways: those that will reside permanently in ROM[5]; 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 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.

Objects that are initialised are usually placed into psects with the name "data" or a name based on "data". Variables that are qualified `near` typically use the psect `rdata`. The PIC data psects use names like `rdata_0` to indicates that they are "near" (there is no `near` qualifier on the PIC - essentially all PIC objects are `near` by default) and the digit indicates a bank number.

Uninitialised objects are placed in psects whose name is "bss" or a name based on "bss". Again, `rbss` would indicate uninitialised objects that are `near`. The PIC compiler uses names like `rbss_0`, where the digit is a bank number. The abreviation "bss" stands for **b**lock **s**tarted by **s**ymbol and was a assembler pseudo-op used in *IBM* systems back in the days when computers were coal-fired. The continued usage of this term is still appropriate as there are some similarities in the way these schemes worked.

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

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. A 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 XA compiler handles these two objects. Other compilers may produce differently structured code. The **PSECT** directive flags are discussed 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 are handled differently by the compilers. They may be allocated space dynamically (for example on the stack) in which case they are not stored in psects by the compiler. Compilers or memory models which do not use a hardware stack, use a *compiled stack* which is an area of memory set aside for the auto and parameter objects for each function. These object will be positioned in a psect. The psect in which they are allocated is defined by a **FNCONF** directive which is placed in the run-time startup code.

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 will be the same, as they 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. They are assigned a load address, and you will see it listed in the map file, but it is not used. The compiler usually 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 assembler[6] 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` 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 be called and hence several temporary areas created. To get around this problem, the psects which contain the directives to

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

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.

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` itself 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` option[7] 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.

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.

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 (i.e. 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 bytes long. These psects are to be placed in ROM and all have the simple definition generated by the code generator:

```
PSECT text,class=CODE
```

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 contiguous[8] and form one larger `text` psect. The following `-p` linker option could be used to position the `text` psect at the bottom of ROM.

```
-ptext=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 an FFFFh, and the remaining locations are for peripheral interrupt vectors. The run-time code usually defines the reset vector using code like the following.

```
GLOBALstart
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

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.

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.

```
GLOBAL_timer_isr
PSECT vectors,ovlrd
ORG   10h
DW    _timer_isr
```

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 is 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.) Other compilers may also operate this way if ROM and RAM are in different memory spaces. 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 for the PIC compiler will be similar to `rdata_0`; those in ROM are like `idata_0`. The digit refers to a PIC 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 class 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 -ROM*ranges* 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 the 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 are 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 43.

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 psects, 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 you 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 there nature is not required to 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 XA. 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 code. 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 -pvectors=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.

Using HPDXA

3.1 Introduction

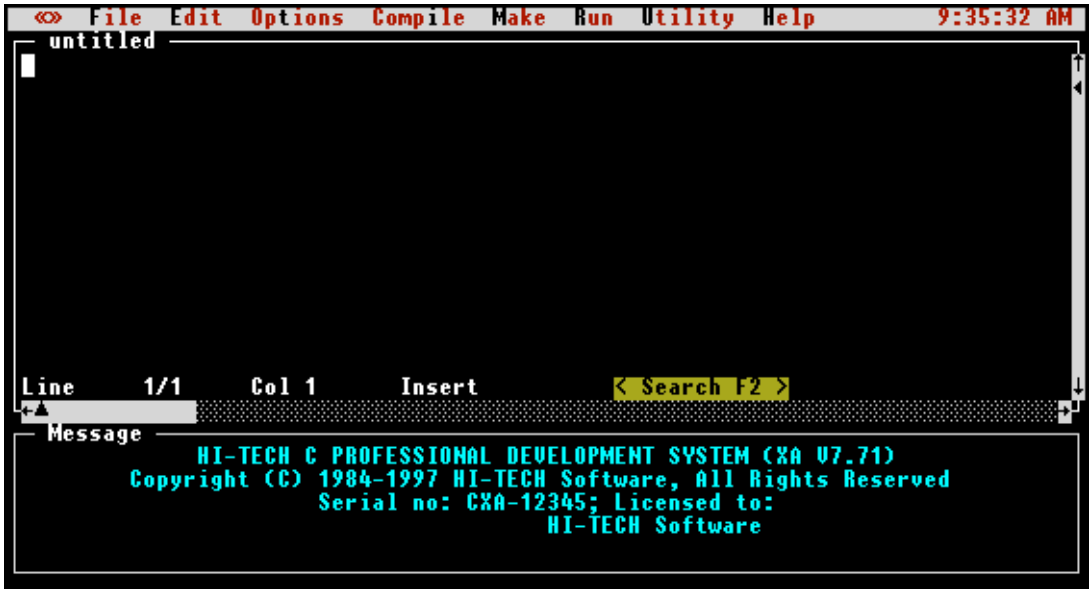
This chapter covers HPD, the **H**I-TECH C **P**rogrammer's **D**evelopment system integrated environment. It assumes that you have already installed your HI-TECH C compiler. If you haven't installed your compiler go to the *Quick Start Guide* and follow the installation instructions there.

HPDXA is the version of HPD applicable to the XA compiler.

3.1.1 Starting HPDXA

To start HPDXA, simply type hpdxa at the 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 on page 49.

Figure 3 - 1 HPDXA Startup Screen



The initial HPDXA screen is broken into three windows. The top window contains the menu bar, the middle window the HPDXA 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.

HPDXA 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, HPDXA 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, HPDXA 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, HPDXA 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 HPDXA is invoked with the command:

```
hpdxa 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 HPDXA 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 card 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 HPDXA 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 HPDXA 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 HPDXA 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 HPDXA exits. The selected number of lines will be saved in the hpdxa.ini file and used for subsequent invocations of HPDXA unless overridden by another /screen option.

HPDXA will recognize and use any mouse driver which supports the standard INT 33H interface. Almost all modern mouse drivers support the 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, HPDXA will still work with the mouse, but the *Mouse Setup* dialog in the <<>> menu will not work.

3.2.3 Colours

Colours are used in two ways in HPDXA. 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 52 shows the colours and their values, whilst Table 3 - 2 on page 52 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 53 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 hpdxa.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 .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.

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

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 53.

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

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

current context of the application. For example, the **Save project** item will not be selectable if no project has been loaded or defined.

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

When browsing through the menu system you will notice that some menu items have *hot key* sequences displayed. For example, the HPDXA 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 HPDXA `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 HPDXA key equivalents is shown in Table 3 - 5 on page 55.

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 rearmost window is brought to the front and each other window on screen shuffles 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 region[1]. 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

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

Table 3 - 5 HPDXA 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

example, the HPDXA compiler error window will not be made front most if it is clicked. Instead it will 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 and 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 HPDXA 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 you should 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 on page 56.

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 menu system activation 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 be 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.

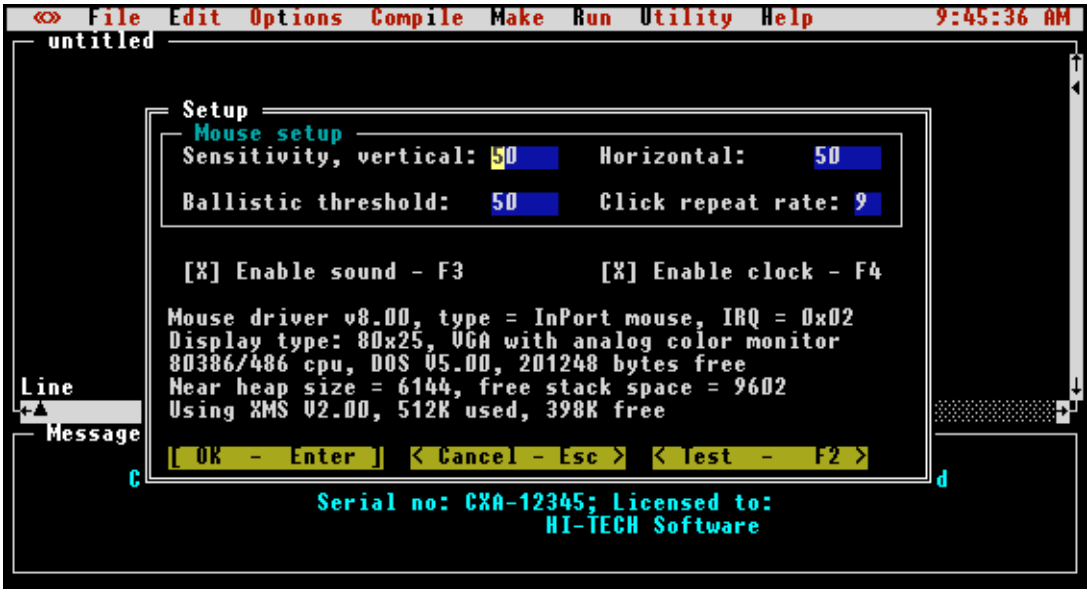
3.2.7 Buttons

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 `HPDXA` error window (Figure 3 - 6 on page 62) 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 HPDXA* entry, which displays information about the version number of `HPDXA`; and the *Setup* entry. Selecting the Setup entry opens a dialog box as shown in Figure 3 - 2 on page 58. This box displays information about `HPDXA'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 `HPDXA's` initialisation file, so they will be reloaded next time you run `HPDXA`.The sound and clock settings will 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 HPDXA. It does not attempt to give you a comprehensive tour of HPDXA'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 HPDXA, 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> hpdxa
```

You will be presented with the HPDXA 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 HPDXA. This will naturally be the infamous “hello world”.

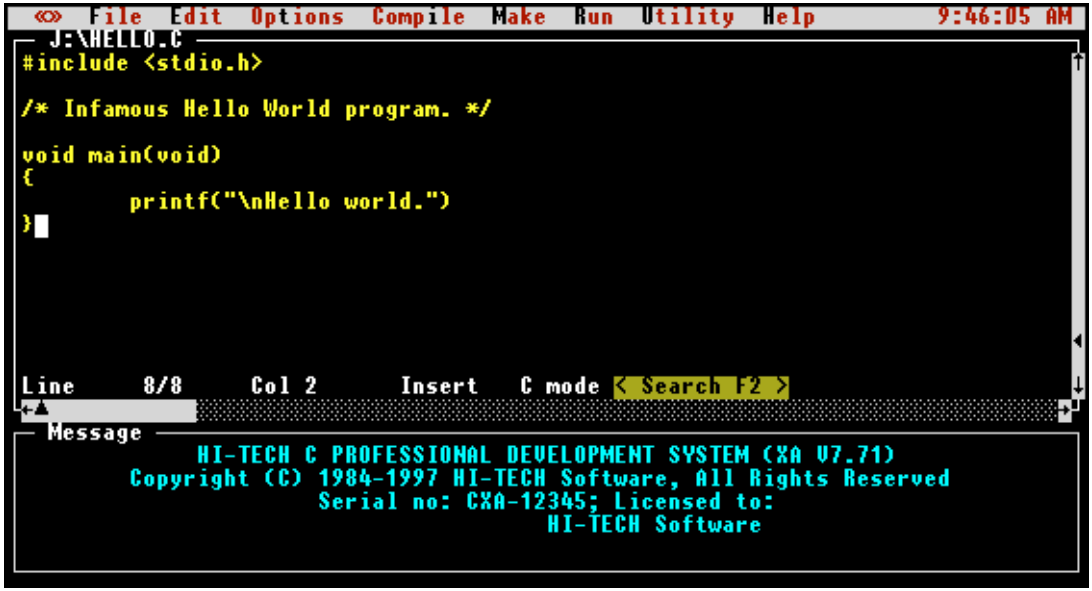
Type the following text, pressing enter once at the end of each line. You can enter blank lines by pressing enter without typing any text.

```
#include  <stdio.h>

main()
{
        printf("Hello, world")
}
```

Note that a semi-colon has been deliberately omitted from the end of the **`printf()`** statement in order to demonstrate HPDXA's error handling facilities. Figure 3 - 3 on page 59, shows the screen as it should appear after entry of the “Hello world” program).

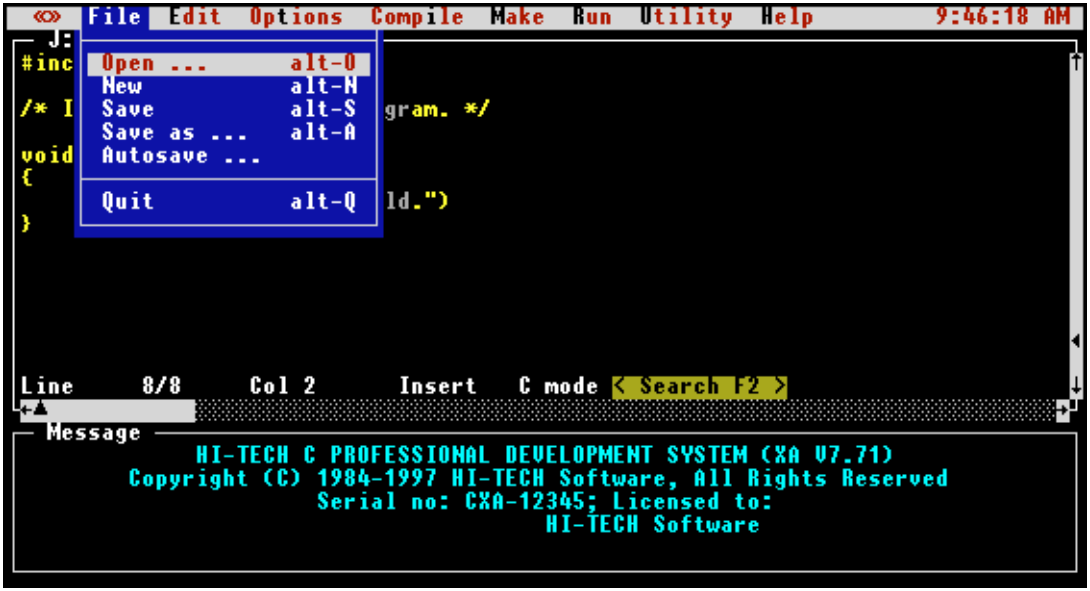
Figure 3 - 3 Hello program in HPDXA



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 60

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

Figure 3 - 4 HPDXA File Menu



- ❒ Open the menu system by pressing `alt-space`
- ❒ Move to the **Edit** menu using the `right arrow` key
- ❒ 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 on page 60) 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.

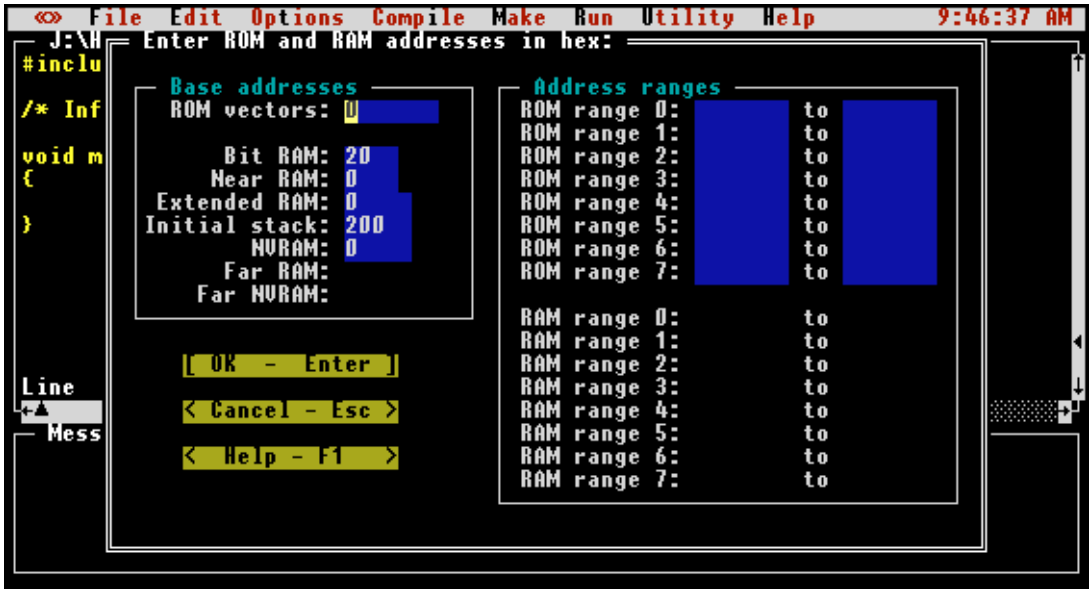
After **Save** has been selected, you should be presented with a *dialog* prompting you for the file name. If HPDXA needs more information, such as a file name, before it is able to act on a command, it will always prompt you with a standard dialog like the one below.

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 hello.c. Type hello.c and then press enter. There should be a brief period of disk activity as HPDXA saves the file.

You need to set the memory configuration. To do this select **ROM & RAM addresses...** from the **Options** menu. You should use 0 for ROM vectors, 20 for Bit RAM and 0 for Near RAM. Set Extended RAM to 0 to concatenate it with Near RAM. Use 200 for Initial Stack, which is used as the initial stack pointer. If you are not using non volatile RAM or all your RAM is non volatile set the NVRAM to 0. Once you have set the memory configuration the dialog should look like Figure 3 - 5 on page 61.

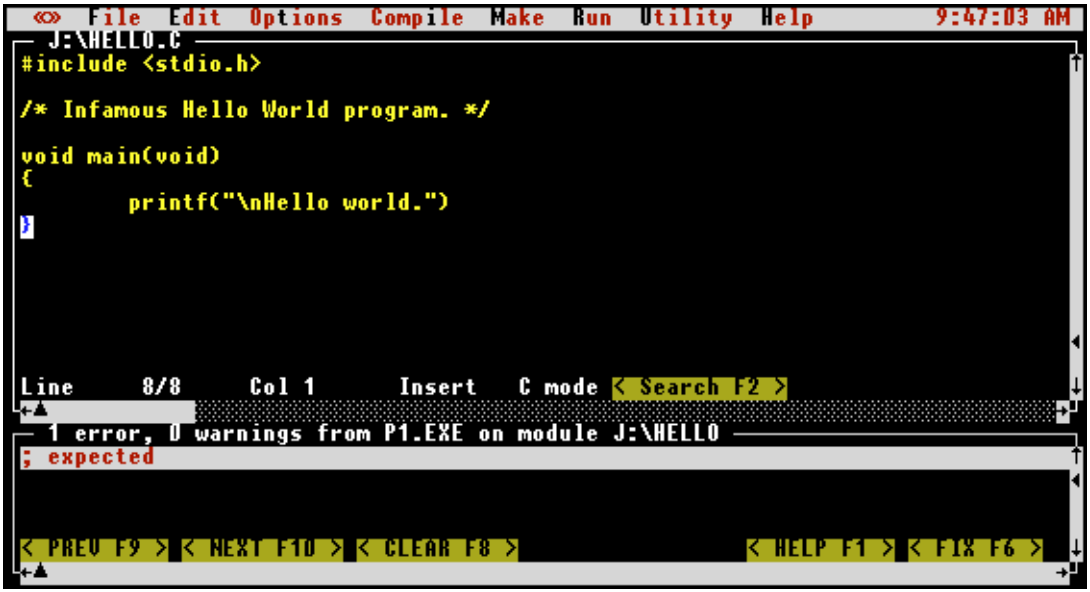
Figure 3 - 5 ROM and RAM Address dialog



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.

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 HPDXA handles compiler errors. After a couple of seconds of disk activity as the CPP and P1 phases of the compiler run, you should hear a "splat" noise. The message window will be replaced by a window containing a number of buttons and the message *; expected*, as shown in Figure 3 - 6 on page 62.

Figure 3 - 6 Error window



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 cgxx.exe. cpp.exe and hlink.exe can also return errors.

In this case, the error window frame contains the message 1 error, 0 warnings from p1.exe indicating that pass 1 of the compiler found 1 fatal error. It is possible to configure HPDXA 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 **printf()** statement. 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 much 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 (cgxx), assembler (asxx) 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 HPDXA 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 HPDXA 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, HPDXA will move to the next error.

To manually correct the error, move the cursor to the end of the **printf()** statement 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. HPDXA 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. HPDXA 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 HPDXA menu system, editor and project facility.

3.4 The HPDXA Editor

HPDXA 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 HPDXA editor without further instruction. HPDXA also supports the standard PC keys, and thus should be readily usable by anyone familiar with typical MS-DOS or *Microsoft Windows* editors.

The HPDXA 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 65, for a list of the valid *ctrl*-K commands.
- WordStar *ctrl-k* command. See the section Keyboard Commands on page 65, 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 65, for a list of the valid *ctrl*-Q commands.
- *Insert* indicates whether 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 - 8 on page 67 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 HPDXA menu bar may still be accessed either by pressing the alt key or by using the middle button on a three button mouse.

3.4.5 Block commands

In addition to the movement and editing command listed in the "Editor Keys" table, the HPDXA 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 - 7 on page 66 lists the *WordStar* compatible block operations which are available.

Table 3 - 7 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

Table 3 - 8 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	*crtl*-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	

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 HPDXA 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 HPDXA 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 HPDXA menus

This section presents a item-by-item description of each of the HPDXA menus. The description of each menu includes a screen print showing the appearance of the menu within a typical HPDXA 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 HPDXA ...
The About HPDXA 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 58

3.5.2 File menu

The **File** menu contains file handling commands,the HPDXA **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.

Save **alt-S**
This command saves the current edited 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 not zero, then the current edit file will automatically be saved to a temporary file at intervals. Should HPDXA not exit normally, e.g. if your computer suffers a power failure, the next time you run HPDXA, it will automatically restore the saved version of the file.

Quit **alt-Q**
The **Quit** command is used to exit from HPDXA 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.

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.

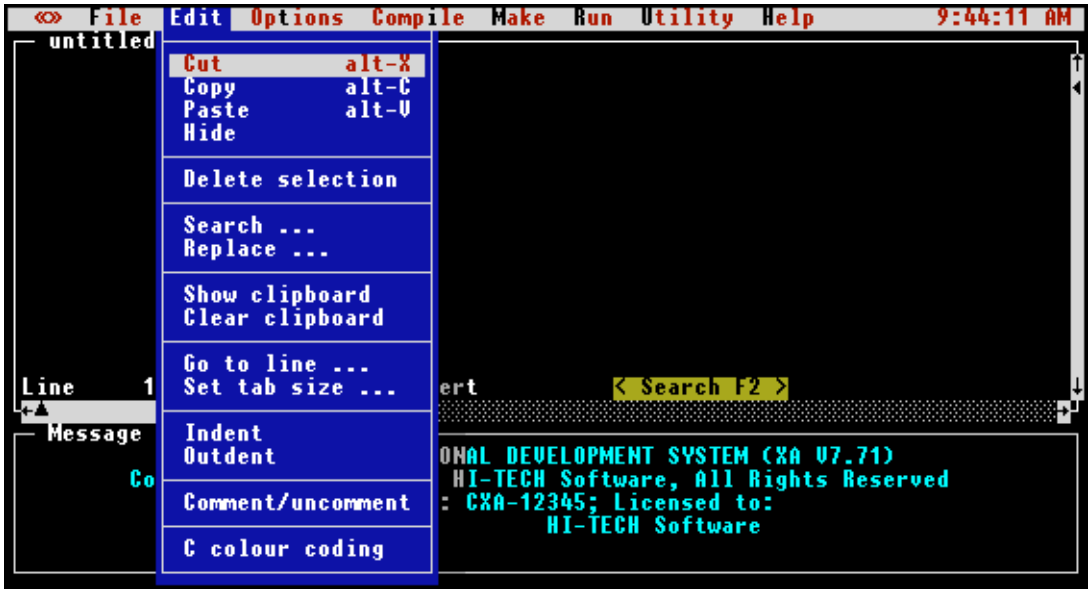
Pick list **ctrl-F4**

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 Table 3 - 7 on page 72.

Figure 3 - 7 HPDXA Edit Menu



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.

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 options 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 HPDXA and how to select specific schemes, you should refer to Colours on page 51.

3.5.4 Options menu

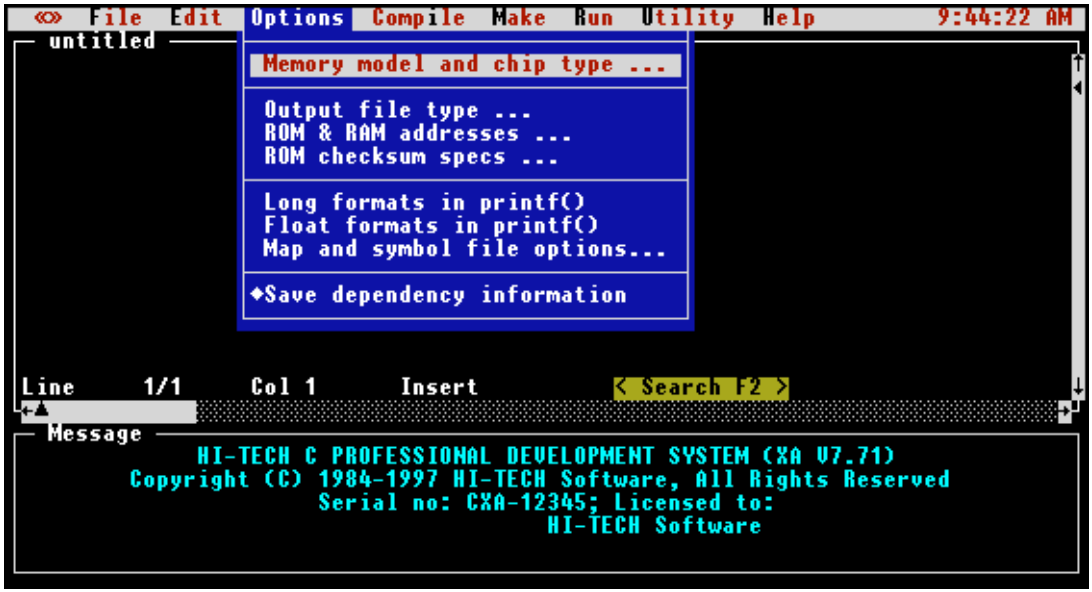
The **Options** menu contains commands which allow selection of compiler options, memory models, and target processor. Selections made in this menu will be stored in a project file, if one is being used. The Options menu is shown in Figure 3 - 8 on page 75..

Memory model and chip type...
This option activates a dialog box which allows you to select the memory model you wish to use. Available memory models are *small*, *medium*, *large* and *huge*. At the time of writing there was only one XA chip type, but as others become available this option will allow you to select the appropriate type. The dialog box also allows you to choose the type of floating point mathematics you wish to use. Available options are *single precision*, *double precision* and *fast double precision.*

Small memory model supports up to 64K code, but only internal RAM (512-1K bytes). Initialized data and strings are stored in ROM and accessed from ROM with move instructions. This leaves the maximum RAM for variables and stack. This is the default setting.

Medium memory model supports 64K ROM and 64K RAM. Initialized data and strings are copied from ROM into RAM at startup time. All data is accessed with indirect addressing, unless the variable has been declared “near”. In that case it will be allocated in internal RAM and accessed with direct addressing. The processor is set into page zero mode by the startup code, thus forcing 16 bit addressing.

Figure 3 - 8 Options Menu



Large memory model is similar to the medium model, except that the processor is operated in normal mode, rather than page zero mode. This means that calls and returns use 24 bit addresses. This allows access to up to 16 Mbyte, depending on the particular chip variant, of code. Data is still limited to 64K bytes, and a single function may not exceed 64K bytes in size. More than 64K of data may be accessed in any model by use of the `far` keyword.

Huge memory model is similar to the large model, except that all pointers are 32 bits, and include information to distinguish between code and data memory.

Single precision floating point uses 32 bit IEEE floating point format for both *float* and *double* data types.

Double precision floating point uses 64 bit IEEE floating point format for *double* data types and 32 bit IEEE floating point format for *float* data types.

Fast double precision floating point uses fast 64 bit (non-IEEE) floating point format for *double* data types and 32 bit IEEE floating point format for *float* data types.

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

ROM & RAM addresses ...
This option allows you to enter the addresses of the ROM and RAM in your target system. For standard values refer to the instructions in the tutorial section on page 62.

The information entered into this dialog box is used by the linker to assign addresses to your code. Note that some of the RAM addresses may be set to zero to allow `HPDXA` to set them automatically. The addresses are used as follows:

- **ROM vectors:** This is the location of the reset vector - this will always be zero, unless you are compiling to run with a debugger, in which case it may be set to a higher value, because the debugger will “reflect” the vectors to a higher address.
- **Bit RAM:** Sets the starting address used to allocate bit variables. Usually you will leave this at 20, since this is the lowest addressable bit location in direct RAM (corresponds to bit no. 100h)
- **Near RAM:** This sets the lowest address used to allocate variables in the internal (directly addressable) RAM. IF this value is set to zero, then the near RAM area will start immediately after the last location used for bit variables.
- **Extended RAM:** In medium and higher models, most variables are allocated in extended RAM, which is not limited to 400h as the near RAM is. This address is the lowest extended RAM address. If set to zero, this will start after the last location used in the near RAM (which may mean some “extended” RAM is actually in the internal RAM area - this is not important).
- **Initial stack:** This address is the value that will be loaded into the stack pointer at startup. The stack will grow downwards from here, so it should be set to a high enough value to leave sufficient space between the stack and the highest allocated variable in RAM. In small model this will usually be set to 200, being the limit of on-board RAM in the XA-G3. The range of the initial stack is 20h to 10000h.
- **NVRAM:** Variables qualified `persistent' will be allocated starting at this address. If left at zero, then the persistent area will be concatenated with the extended RAM.
- **Far RAM:** In large and huge models you can use `far' variables. This is the lowest address to assign to these variables. It defaults to 10000h (64K) but can be set lower. If the RAM ranges are set this value will not be used.
- **Far NVRAM:** Provides the location for placing variables qualified `far persistent'. If set to 0 this will be concatenated with the other far RAM.

Address ranges are used as follows:

- **ROM Ranges:** If these are not set, then the compiler will use ROM from the 'ROM vectors address' to the end of the memory space (16Mbytes in large and huge models, 64K in small and medium).
- If set, each range should represent a range of addresses that the compiler can allocate code into. This can be used if you have non-contiguous ROM, or need to leave space for checksums etc. In the small and medium memory model, these ranges must be below 64K.
- **RAM Ranges:** If these are not set, then the compiler will use RAM from the 'far RAM' to the end of the memory space. If set, each range should represent a range of addresses that the compiler can allocated far data into. These ranges are not available in small memory model since it does not support far data.

ROM checksum specs ...
HPDXA allows you to enter several specifications for checksums to be embedded in your ROM. Each checksum is calculated by summing each byte in a range, then adding a constant. The resultant value is stored as 1, 2 or 4 bytes at a specified address. 2 and 4 byte checksums can be stored in high-to-low (default) or low-to-high order.

The "Copy Ranges" button will copy the ROM Ranges (set in the ROM and RAM Addresses dialog) into the checksum ranges. Ensure that any locations into which you will store checksums are not included in any ROM range. For example, a typical setup might be a ROM range of 0000-7FFE with a 1 byte checksum at 7FFF. Set this up in the ROM and RAM Addresses multi-ROM dialog with ROM range 0 as 0000-7FFE. Then in the Checksum Ranges dialog simply clicking Copy Ranges will set up the checksum range 0 to sum from 0 to 7FFE and store the result at 7FFF.

Long formats in printf()
This option is used to tell the linker that you wish to use the **`long printf()`** support library. The long library includes a version of **`printf()`** which supports the long output formats %1d, %1u and %1x. If you use this option the compiled application will increase in size. You should only use this option if you want to use long formats in **`printf()`**, it is not necessary if you merely want to perform long integer calculations. If you select this option a marker appears beside the line in the menu.

Float formats in printf()
This option is used to tell the linker that you wish to use the **`floating point printf()`** support library. The float library includes a version of **`printf()`** which supports the floating point output formats %e, %f and %g. If you use this option the compiled application will be larger. It is not required to perform floating point calculations, so only use it if you wish to use floating point formats in **`printf()`**.

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 a HI-TECH Software debugger like `LUCIFER`, of an in-circuit emulator, 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.

Avocet format symbol file
Use this option to select generation of an Avocet AVSIM compatible symbol file.

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

3.5.5 Compile menu

The **Compile** menu, shown in ,Figure 3 - 9 on page 79 contains the various forms of the compile command along with several machine independent compiler configuration options.

Compile and link **F3**
This command will compile a single source file and then invoke the linker and `objtohex.exe` 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 `hello.c` would be compiled to `hello.exe`

Compile to .OBJ **shift-F3**
Compiles a single source file to a `.obj` file only. The linker and `objto.exe` are not invoked. `.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.

Figure 3 - 9 HPDXA Compile Menu

Preprocess only to .PRE
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 `XAC` command.

Optimization ... **alt-Z**
Selecting this item will open a dialog allowing you to select different kinds and levels of optimization. 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 **`near`**, **`far`** and **`interrupt`** which are used to allow access to the direct addressing mode under programmer control, and to handle interrupts using C code. If this option is used, these keywords are changed to **`__near`**, **`__far`** and **`__interrupt`** respectively so as to strictly conform to the ANSI standard. This option also disables the bit data type. It's the same as the `-STRICT` option to `XAC`.

Pre-process assembler files
Selecting this item will make `HPDXA` 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 `test.c` will produce a listing file called `test.lst`.

Generate C source listings
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.

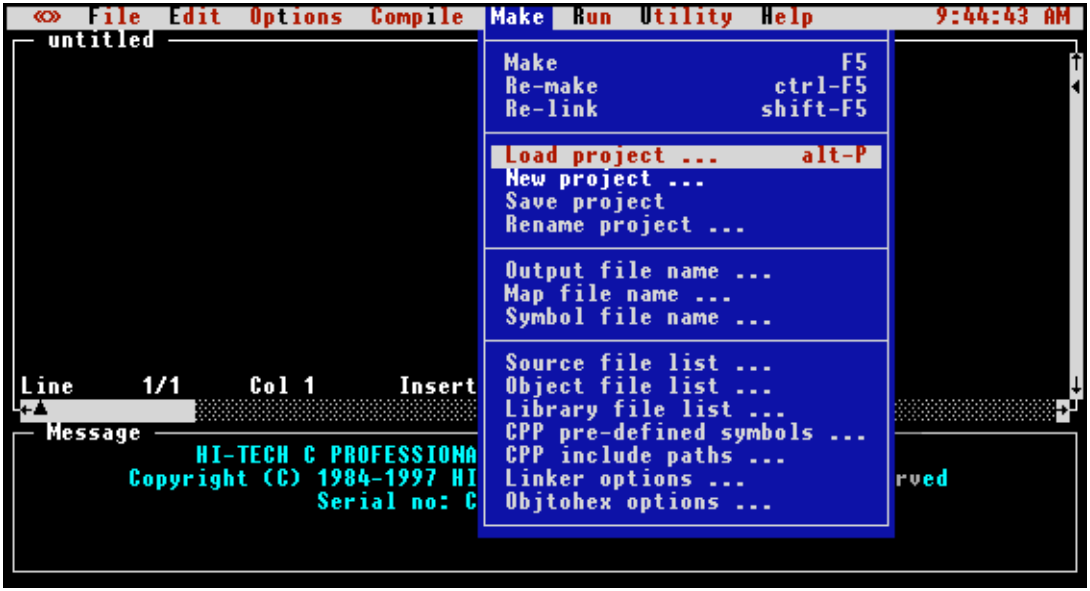
You can only generate *either* a C source listing *or* an assembler listing.

3.5.6 Make menu

The **Make** menu (Figure 3 - 10 on page 81) contains all of the commands required to use the `HPDXA` *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, `HPDXA` will present several dialogs to allow you to set up the memory model.

Figure 3 - 10 HPDXA Make Menu



- ❒ 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.

Make **F5**

The Make command re-compiles the current project. When Make is selected, HPDXA re-compiles any source files which have been modified since the last Make command was issued. HPDXA 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 `.obj` files are current but the output file cannot be found, HPDXA 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, HPDXA will print a message in the message window indicating that nothing was done.

HPDXA 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, i.e. if you modify a header file, any source files including that header file will need to be recompiled.

If you forget to use the source file list to select the files to be included, HPDXA 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 `.obj` files and then selecting **Make**.

Re-link **shift-F5**
The Re-link command relinks the current project. Any `.obj` 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 HPDXA will present several dialogs to allow you to configure the project. These dialogs will allow you to select: processor type, memory model and floating point type; output file type; ROM and RAM addresses; optimization settings; and map and symbol file options. You will still need to enter source file names in 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 an `.exe` file is being generated, the output file name will be automatically set to `prog1.exe`.

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. If a mark character appears against this item, map file generation has been selected. 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. `prog1.sym`. The symbol file produced is suitable for use with any HI-TECH Software debugger.

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 `HPDXA` 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 module for the current code generation model. For example, if a project is generating small model code, by default this list will contain the small model run-time module `rtxa--s.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 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 libraries for the memory model being used. For example, if the current project is generating small model code and floating point code is in use, this list will contain the libraries `xa--sc.lib` and `xa--sf.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, floating point library, standard C library. The floating point library should be linked before the standard C library

if floating point is being used. 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 XAC. 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, for example **`<stdio.h>`** is included, 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 `HPDXA` to be modified. The default contents of the linker command line are generated by the compiler from information selected in the Options menu: memory model, etc. **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 `HPDXA` to be modified. Normally you will not need to change these options as the generation of binary files and HEX 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 - 11 on page 85 contains options allowing MS-DOS commands and user programs to be executed. It also contains the options to allow you to run code using the `LUCIFER` debugger.

DOS command ... **alt-D**
This option allows a DOS command to be executed exactly like it had been entered at the COMMAND.COM prompt. This command could be an internal DOS command like DIR, or the name of a program to be executed. If you want to escape to the 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 a DOS COMMAND.COM shell, i.e. you will be immediately presented with a DOS prompt, unlike the DOS command item which prompts for a command. To return to `HPDXA`, type “exit” at the DOS prompt.

Figure 3 - 11 HPDXA Run Menu

Download ...
This option runs the LUCIFER debugger, automatically downloads the current output file and loads the current symbol file. If the debugger has not been set up, this option will be unavailable. You should use the *Debugger setup ...* command instead. LUCIFER can only download *Motorola* HEX, *Intel* HEX and Binary type files. You should not attempt to download any other file types.

Debugger ...
This option runs the LUCIFER debugger with the current symbol file. This option does not download user code, so can be used to return to a suspended LUCIFER session.

Debugger setup ...
This option activates a dialog box which allows you to select the serial port parameters for the LUCIFER debugger. Once you have set up the parameters they are saved in the project file with the other settings. The default settings for the XA version of LUCIFER are saved in LUCXA_ARGS. For more information refer to the Lucifer source level debugger on page 205.

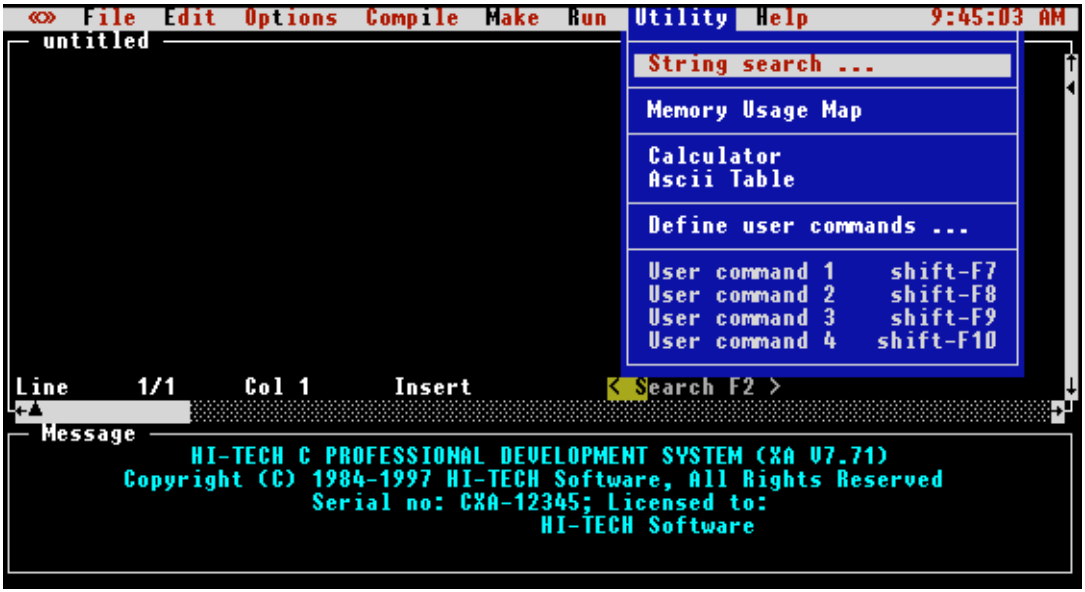
Auto download after compile
This option allows you to enable or disable automatic downloading of code after compiling. When it is enabled and the LUCIFER debugger set up, at successful compilation HPDXA automatically invokes

LUCIFER to download the current output file and load the current symbol file. When it is enabled a mark appears before the item

3.5.8 Utility menu

The **Utility** menu (Figure 3 - 12 on page 86) contains any useful utilities which have been included in HPDXA.

Figure 3 - 12 HPDXA Utility 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 also 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 Escape 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.

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 `Escape` 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 `Escape` while the ASCII table is the front most window.

Define user commands...

Table 3 - 9 Macros usable in user commands

Macro name	Meaning
$(LIB)	Expands to the name of the system library file directory; eg C:\HPDXA\LIB\
$(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 AUDIO.PRJ, this macro will expand to AUDIO with no dot or file type.

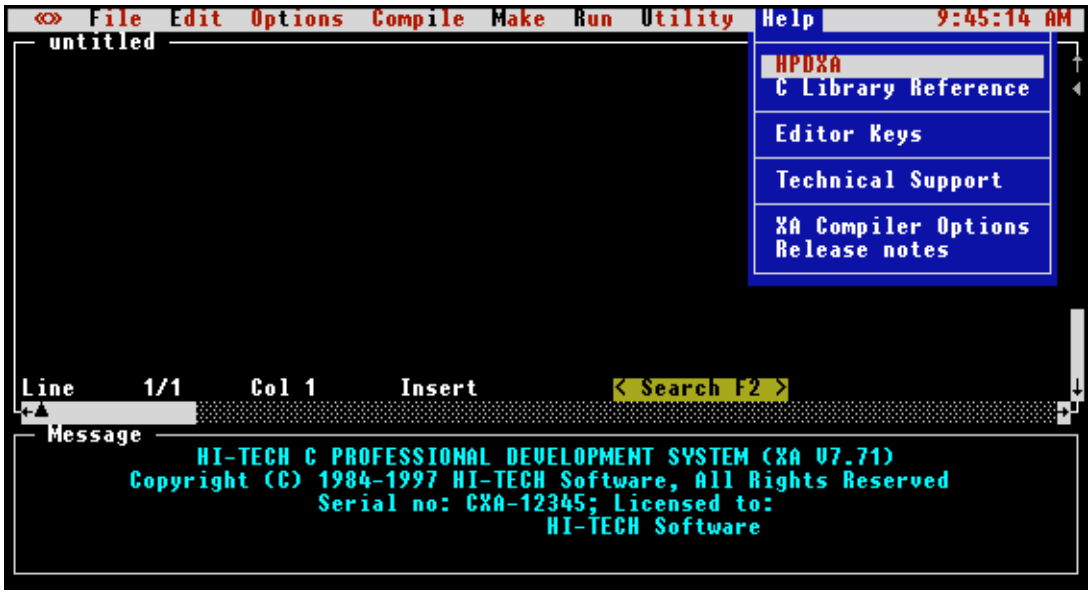
In the Utility menu are four user-definable commands. This item will invoke a dialog box which will allow you to define those commands. By default the commands are dimmed (not selectable) but will be enabled when a command is defined. Each command is in the form of a DOS command, with macro substitutions available. The macros available are listed in Table 3 - 9 on page 87. Each user-defined

command has a hot key associated. They are shift F7 through shift F10, for commands 1 to 4. When a 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 HPDXA.

3.5.9 Help menu

The **Help** menu (Figure 3 - 13 on page 88) contains items allowing you to obtain help about any topics listed.

Figure 3 - 13 HPDXA Help Menu



On startup, HPDXA 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_XX_HLP. If this is not set, it will be derived from the full path name used when HPDXA 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.

HPDXA
This option produces a window showing all the topics for which help is available. These include Checksum specifications, compiler optimizations, editor searching, memory models and chip types, ROM and RAM addresses and string search.

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`.

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 **`printf()`** entry and double click on the reference to **`fprintf()`**, you will be taken to the entry for **`fprintf()`**.

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.

XA Compiler options
This option displays a window showing all the XA 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.

XAC Command Line Compiler Driver

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

```
XAC [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 104). Source files, object files and library files are distinguished by XAC solely by the file type or extension. Recognised 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 XAC File Types

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

XAC 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 XAC 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 XAC with a -C option to compile several source files to object files, then to create the final program, invoke XAC 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 XAC with a long list of options and files you may create a command file containing the XAC command line, and invoke XAC with its

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

```
-V -O -Otest.hex -A0,20,1E0,0,0,0 -Bm \
file1.obj file2.obj mylib.lib -Lf
```

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

```
XAC < xyz.cmd
```

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

In a batch file, a *space* character followed by 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

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

The standard library contains a version of **printf()** that supports only integer length values. If you want to print long values with **printf()**, or **sprintf()** or related functions, you must specify a -Ll option. This will search the library containing the long version of **printf()**. For floating-point and long **printf()** support, use the -Lf option which will search the library containing the floating-point version of **printf()**. You do not need the -Ll option if you have specified the -Lf option.

4.3 Standard Run-Time Code

XAC will also automatically provide the standard run-time module appropriate for the memory model. If you require any special powerup initialisation, rather than replace or modify the standard run-time module, you should use the *powerup* routine feature (see page 117).

4.4 XAC Compiler Options

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

4.4.1 -A*spec*: Set ROM and RAM Addresses

The -A option is used to set the ROM and RAM addresses which will be used to link your code to absolute addresses. This option takes the form:

-Arom,bitram,stack,intram,extram,nvram,farram,farnvram

where:

Table 4 - 2 XAC Options

Option	Meaning
-A*spec*	Specify memory addresses for linking
-AAHEX	Generate an American Automation symbolic HEX file
-ASMLIST	Generate assembler .LST file for each compilation
-AV	Select AVOCET format symbol table
-BIN	Generate a Binary output file
-Bh	Select *huge* memory model
-Bl	Select *large* memory model
-Bm	Select *medium* memory model
-Bs	Select *small* medium model
-C	Compile to object files only
-CLIST	Generate C source listing file
-CR*file*	Generate cross-reference listing
-D*macro*	Define preprocessor macro on command line
-DOUBLE	Use 64 bit floating-point for *double*
-E*[[+]file]*	Define format for compiler errors / redirect errors
-FDOUBLE	Use fast double float format
-G*file*	Generate source-level symbol table
-H*file*	Generate enhanced symbol table
-HELP	Print summary of options
-INTEL	Generate an Intel HEX format output file
-I*path*	Specify a directory pathname for include files
-L*library*	Specify a library to be scanned by the linker
-L-*option*	Specify -*option* to be passed directly to the linker
-M*file*	Request generation of a MAP file
-MOTOROLA	Generate a Motorola HEX format output file
-N*length*	Set identifier length to *length* (default is 31 characters)
-O	Enable post-pass optimization
-O*file*	Specify output filename and type
-OMF51	Produce an OMF-51 output file
-P	Preprocess assembler files
-PRE	Produce preprocessed source files
-PROTO	Generate function prototype information
-PSECTMAP	Display complete memory segment usage after linking

Table 4 - 2 XAC Options

Option	Meaning
-q	Specify quiet mode
-ROM*ranges*	Specify ROM ranges for code
-RAM*ranges*	Specify RAM ranges for far data
-S	Compile to assembler source files only
-SA	Compile to Avocet AVMAC assembler source files
-STRICT	Enable strict ANSI keyword conformance
-TEK	Generate a Tektronix HEX format output file
-U*macro*	Undefine a predefined macro
-UBROF	Generate an UBROF format output file
-UNSIGNED	Make default character type *unsigned*
-V	Display compiler pass command lines
-W*level*	Set compiler warning level
-X	Eliminate local symbols from symbol table
-Zg	Enable global optimization in the code generator

rom is the address in ROM where program code is to start. For most XA variants this will be address 0, the lowest ROM address available. If compiling code which is to be downloaded into RAM using the *Lucifer* debugger, this address will be the start of the downloadable RAM area. The ROM area starts with the reset vector, followed by any other interrupt vectors which have been initialized using the interrupt vector handling macros defined in header file `<intrpt.h>`.

bitram is the starting address of RAM to be used for **`bit`** variables. This value should be between 20h and 3Fh, as this area is standard bit-addressable RAM. The default value for *`bitram`* is 20h (which corresponds to bit address 100h). **`Bit`** variables will be allocated at this address. Other variables will be allocated after the bit variables unless otherwise specified via *`intram`*.

stack is the value to be used as the initial stack pointer. This would usually be 200 for the XA-G3 with internal RAM only, as the on-board RAM extends up to 1FFh. If using external memory, this will usually be the top of external memory in page 0. The stack value may range from 20h to 10000h.

intram The internal RAM address to be used for allocation of **`near`** variables (or all variables in small model). If set to 0 or omitted, then these variables will be concatenated with the bit RAM (i.e. *`bitram`*).

extram is the start of the external RAM area to be used (in medium and higher models). Variables not qualified **near** will be allocated here. If omitted or set to zero, the end of the internal RAM area will be used instead.

nvram is the address of a block of non-volatile RAM to be used for **persistent** variables. If all RAM is non-volatile or **persistent** variables are not used, this value should be zero. This will cause the non-volatile RAM area to be allocated from within the standard RAM area. The default value is zero.

farram is the address of a block of external RAM that will be used for allocating **far** variables. This should be the base of one of the extended RAM banks. The default value is 10000h, i.e. the second 64k bank of RAM. If **far** variables are not used, the value of this area is not important. This value may be overridden by the use of the `-RAM` option (see below).

farnvram The address at which to allocate variables qualified both **far** and **persistent**.

All values taken by the `-A` option are hexadecimal (base 16) numbers, and should be specified without a trailing "`H`". Thus, the option

```
-A0,20,200
```

is correct, but the option:

```
-A0H,20H,200H
```

is not. Some examples of valid `-A` options follow:

```
-A0,20
```

Start code (ROM) at 0h; RAM variables start at 20h; no non-volatile or banked RAM specified. Stack start defaults to 200h.

```
-A4000,30,100
```

ROM at 4000h; internal RAM variables start at 30h; initial stack is 100h.

```
-A0,20,1F00,0,0,1FF0,20000
```

ROM at 0h; RAM variables start at 20h; initial stack is 1F00h; RAM variables are placed after the RAM bit variables; external RAM is concatenated with the internal RAM; the RAM extends up to 1FF0h, from which address the non-volatile RAM variables will be placed; far (i.e. external) RAM starts at 20000h (128k).

4.4.2 -AAHEX: Generate American Automation Symbolic HEX

The -AAHEX option tells XAC 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.

4.4.3 -ASMLIST: Generate Assembler .LST Files

The -ASMLIST option tells XAC to generate an assembler .lst file for each compilation.

4.4.4 -AV: Select Avocet Symbol File

The -AV option is used in conjunction with the -H option to generate *Avocet* style symbol tables for use with AVSIM simulators and certain in-circuit emulators. This option only sets the symbol table format, it does not tell the compiler to actually generate a symbol file. In order to generate an *Avocet* symbol file you should use the -AV option with the -H option. For example:

```
XAC -AV -Htest.sym -A0,30,100 test.c
```

will generate an *Avocet* style symbol table called test.sym. The -AV option should not be used with the XAC -G option as *Avocet* symbol tables make no provision for source-level debug information.

4.4.5 -BIN: Generate Binary Output File

The -BIN option tells XAC 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 option.

4.4.6 -Bh: Select Huge Memory Model

The -Bh option is used to select code generation using the *huge memory model*. The huge model is similar to the large model (see below) but all pointers in huge model are 32 bits wide, and include information to distinguish between code and data memory. This allows a pointer to refer to either code or data memory, whereas in the other models, a pointer is either a pointer into the data space or the code space, this distinction being made at compile rather than run time.

The overhead of using huge model is larger code size and reduced speed. The advantage is that it can make programming easier in some circumstances. The libraries used with huge model are listed in Table 4 - 3 on page 97.

4.4.7 -Bl: Select Large Memory Model

The -Bl option is used to select code generation using the *large memory model*. Large model uses the far call and return features of the XA to allow up to 16MB of code memory to be accessed (depending

Table 4 - 3 Huge Model Libraries

Library	Purpose
rtxa--h.obj	Huge model run-time module
xa--hc.lib	Huge model standard library, 32 bit doubles
xa--hl.lib	Huge model printf library, long support
xa--hf.lib	Huge model printf library, long and float support
xa-dhc.lib	Huge model standard library, 64 bit doubles
xa-dhl.lib	Huge model printf library, long support, 64 bit doubles
xa-dhf.lib	Huge model printf library, float support, 64 bit doubles
xa-fhc.lib	Huge model standard library, fast doubles
xa-fhl.lib	Huge model printf library, long support, fast doubles
xa-fhf.lib	Huge model printf library, float support, fast doubles

on the XA variant). This model is also sometimes referred to as a “banked” model. Code memory is allocated in the ROM space from zero upwards, and will automatically fill additional 64k banks as required. Other than this, the large model is the same as the medium model, i.e. it allows up to 64k data memory, plus additional data memory declared as **far**.

Function addresses in large model are 24 bits wide, but 32 bits is actually allocated where a function pointer is stored in memory. The run-time module and libraries associated with large model are listed in Table 4 - 4 on page 97

Table 4 - 4 Large Model Libraries

Library	Purpose
rtxa--l.obj	Large model run-time module
xa--lc.lib	Large model standard library, 32 bit doubles
xa--ll.lib	Large model printf library, long support
xa--lf.lib	Large model printf library, long and float support
xa-dlc.lib	Large model standard library, 64 bit doubles
xa-dll.lib	Large model printf library, long support, 64 bit doubles
xa-dlf.lib	Large model printf library, float support, 64 bit doubles
xa-flc.lib	Large model standard library, fast doubles
xa-fll.lib	Large model printf library, long support, fast doubles
xa-flf.lib	Large model printf library, float support, fast doubles

4.4.8 -Bm: Select Medium Memory Model

The `-Bm` option is used to select code generation using the *medium memory model*. Medium model allows up to 64k code and 64k data (therefore the **`far`** keyword has *no effect*). Function pointers are 16 bits wide. Access to data variables is via indirect and indexed addressing modes. Variables declared **`near`** will be placed in internal RAM (below 400h) and will be accessed with direct addressing. The run-time module and libraries used with medium model are listed inTable 4 - 5 on page 98.

Table 4 - 5 Medium Model Libraries

Library	Purpose
`rtxa--m.obj`	Medium model run-time module
`xa--mc.lib`	Medium model standard library, 32 bit doubles
`xa--ml.lib`	Medium model printf library, long support
`xa--mf.lib`	Medium model printf library, long and float support
`xa-dmc.lib`	Medium model standard library, 64 bit doubles
`xa-dml.lib`	Medium model printf library, long support, 64 bit doubles
`xa-dmf.lib`	Medium model printf library, float support, 64 bit doubles
`xa-fmc.lib`	Medium model standard library, fast doubles
`xa-fml.lib`	Medium model printf library, long support, fast doubles
`xa-fmf.lib`	Medium model printf library, float support, fast doubles

4.4.9 -Bs: Select Small Memory Model

The `-Bs` option selects the *small* memory model of the XA compiler. This is the default memory model and will be used even if no `-B` option is given. Small model allows up to 64k code, but only up to 1k data. All data is accessed via direct addressing. Statically initialized data and string constants are placed in the ROM and accessed directly from there with **`movc`** instructions. This economises on RAM usage, but it means that statically initialised variables cannot be modified at run time.

The run-time module and libraries for small model are listed inTable 4 - 6 on page 99.

4.4.10 -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:

```
XAC -O -Zg -C main.c module1.c asmcode.as
```

Table 4 - 6 Small Model Libraries

Library	Purpose
rtxa--s.obj	Small Model run-time module
xa--sc.lib	Small Model standard library, 32 bit doubles
xa--sl.lib	Small Model printf library, long support
xa--sf.lib	Small Model printf library, long and float support
xa-dsc.lib	Small Model standard library, 64 bit doubles
xa-dsl.lib	Small Model printf library, long support, 64 bit doubles
xa-dsf.lib	Small Model printf library, float support, 64 bit doubles
xa-fsc.lib	Small Model standard library, fast doubles
xa-fsl.lib	Small Model printf library, long support, fast doubles
xa-fsf.lib	Small Model printf library, float support, fast doubles

The compiler would 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:

```
XAC -A0,20,1E0 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 `-O` option is used to specify an output filename 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`.

4.4.11 -CR*file*: Generate Cross Reference Listing

The `-CR`*`file`* 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`, `XAC` 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 `XAC` command. For example, to generate a cross reference listing which includes the source modules `main.c, module1.c` and `nvram.c,` compile and link using the command:

```
XAC -CRmain.crf main.c module1.c nvram.c
```

4.4.12 -CLIST: Produce C Listing File

This option will generate a listing file for each C source file, containing line numbers with tabs formatted to spaces on 8 character stops. The listing file will be called `file.lst` where `file` is the base name of

the C source file. If the -ASMLIST option is specified, the C source listing file is overwritten by the assembler listing file.

4.4.13 -D*macro*: 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
```

where ***text*** is the textual substitution required. Thus, the command:


```
XAC -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
```

4.4.14 -DOUBLE

By default the compiler will generate code and use libraries that treat **double** and **float** variables as 32-bit IEEE format. The -DOUBLE option will cause **double** variables to be treated as 64-bit IEEE format. This also means that the maths library functions, **printf()**, etc., will use 64-bit floating-point arithmetic. With the -DOUBLE option **float** variables will remain in the 32-bit format.

4.4.15 -E*[[+]file]*: Define Format for Compiler Errors / Redirect 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()
     6: PORT_A = x00;
                 ^ undefined identifier: x00
```

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

4.4.15.1 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 6 7: undefined identifier: x00
```

indicating that the error occurred in file x.c at line 6, offset 7 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 6, column 7.

4.4.15.2 Modifying the Standard -E Format

If the -E option does not meet your editor's requirements, 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 - 7.

Table 4 - 7 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.

4.4.15.3 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:

```
XAC -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:

```
XAC -E x.c >> errlist
```

Some editors do not allow the standard command line redirection facilities to be used when invoking the compiler. To work with these editors, `XAC` allows the error listing filename 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:

```
XAC -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 filename, for example:

```
XAC -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:

```
XAC -Eproject.err -O -Zg -C main.c
XAC -E+project.err -O -Zg -C part1.c
XAC -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
```

4.4.16 -FDOUBLE: Select Fast Double Precision

This option selects fast double precision floating-point format for **`double`** variables. **`Float`** variables are still in IEEE 32-bit format. The fast double format provides 48 bits of precision, with a 15-bit exponent. The arithmetic operations are substantially faster than using IEEE floating-point format.

4.4.17 -G*file*: 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 is called *xxx*`.sym`, where *xxx* is the filename of the first source file on the command line. 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. This option should not be used in conjunction with *Avocet* style symbol tables (`XAC` option `-AV`) as the *Avocet* symbol table format makes no provision for source-level debug information. Note that all source files for which source-level debugging is required should be compiled with the `-G` option, for example:

```
XAC -G -C test.c
XAC -C module1.c
XAC -A0,30,2000 -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.

4.4.18 -H*file*: Generate Assembler Level Symbol File

The `-H` option generates a symbol file without any source-level information. HI-TECH Software debuggers will only be able to perform assembler-level debugging when using symbol files generated using `-H`. Normally the symbol file generated using this option will be a HI-TECH Software format symbol table, however if this option is used in conjunction with the `XAC` option `-AV`, an *Avocet* style symbol table will be generated. *Avocet* symbol tables are used by certain in-circuit emulators.

4.4.19 -HELP: Display Help

The `-HELP` option displays information on the `XAC` options.

4.4.20 -INTEL: Generate INTEL Hex File

The `-INTEL` option directs `XAC` to generate an *Intel* HEX file if producing a file with `.hex` extension. This is the default option, when no file type specifier is used.

4.4.21 -I*path*: 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:

```
XAC -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*.

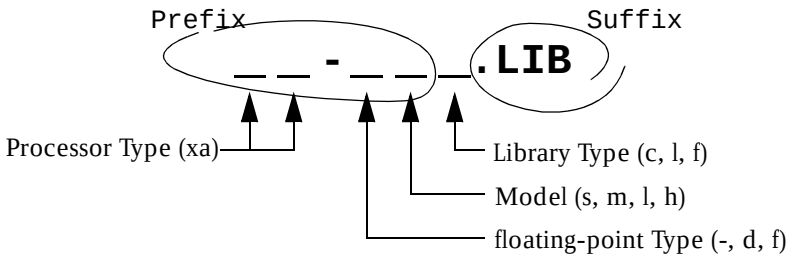
4.4.22 -L*library*: Scan Library

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 alternate versions of standard library functions to be accessed.

For example, if using XA small memory model, the floating-point version of **`printf()`** and **`sprintf()`** can be linked in preference to the standard version by searching the library `xa--sf.lib` using the option `-Lf`.

The argument to `-L` is a library keyword to which the standard library prefix and suffix `.lib` are added. The library prefix depends on which processor and memory model are being used. The standard library prefixes are shown in Figure 4 - 1 on page 104, where: *Processor Type* is "`XA`" for the *Philips* XA; the *floating-point Type* is "`-`" for standard, "`d`" for double precision and "`f`" for fast double; *Model* is "`s`" for small, "`m`" for medium, "`l`" for large and "`h`" for huge; and the *Library Type* is "`c`" for the standard library, "`l`" for long support using, **`printf()`** and **`sprintf()`**, and "`f`" for floats and longs for use with **`printf()`** and **`sprintf()`**.

Figure 4 - 1 Library Prefixes and Suffixes



All libraries must be located in the LIB subdirectory of the compiler installation directory. Libraries in other directories can only be accessed using `HPDXA` or by invoking the linker directly. The complete set of libraries and run-time modules supplied with the compiler is listed in Table 4 - 3 to Table 4 - 6.

4.4.23 -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 XAC. If -L is followed immediately by any text starting with a *dash* "-" character, the text will be passed directly to the linker without being interpreted by XAC.

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 HPDXA 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 **xram**. 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 (xram psect) were to reside at address 1800h, you could use the XAC option -L-Pxram=1800h as follows:

```
XAC -Bl -L-Pxram=1800h -A0,30,8000,4000 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 XAC as -L-N.

4.4.24 -M*file*: 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.

4.4.25 -MOTOROLA: Generate Motorola S-Record HEX File

The -MOTOROLA option tells XAC to generate a *Motorola* S-Record HEX file if producing a file with .hex extension. This option has no effect if used with an option which specifies a .bin file output.

4.4.26 -N*length*: Specify Identifier Significant Length

By default identifiers are truncated at 31 characters, so that two identifiers that were the same in their first 31 characters would be considered identical. This is consistent with the 31 character minimum specified by the ANSI/ISO standard for C. Some applications may require identifiers to be distinguished by more than 31 characters. This option sets the significant number of characters to *length*. It may not be set outside the bounds 31-255. Use this option cautiously as use of identifiers longer than 31 characters will mean the code may not compile with other ANSI-conformant compilers.

4.4.27 -O: Invoke Optimizer

-O invokes the peephole optimizer after the code generation pass. Peephole optimization reduces the code size by removing redundant jump and register load instructions.

4.4.28 -O*file*: 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:

```
XAC -Otest.bin -A0,30,2000 prog1.c part2.c
```

will produce a binary filenamed `test.bin`.

4.4.29 -OMF51: Produce OMF-51 Output File

This option will make the compiler generate an output,`.omf`, file in *Intel* OMF-51 (strictly speaking, AOMF-51) format, along with generating the default *Intel* `.hex` file. This format is used by some in-circuit emulators. It supports only a 64k address space each for code and data.

4.4.30 -P: Preprocess Assembly Files

`-P` causes XAC to preprocess assembler files before they are assembled.

4.4.31 -PRE: Produce preprocessed Source Code

`-PRE` is used to generate preprocessed C source files with an extension `.pre`.

4.4.32 -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, `.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>
int
add(arg1, arg2)
int *   arg1;
int *   arg2;
{
        return *arg1 + *arg2;
}
```

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

If compiled with the command:

```
XAC -PROTO test.c
```

XAC 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 */
```

4.4.33 -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
--------|-------------------------|-----------------------------
vectors | Interrupt vectors       |  0000H -  007FH
const   | 'const' class data      |  0800H -  081DH
```

```
code    | 'code' class data        |  081EH -  091FH
strings | Unnamed string constants |  0920H -  0979H
data    | Initialized XDATA vars   |  097AH -  09ABH
text    | Program and library code |  09ACH -  13D9H

Memory Usage Map:

 CODE:   0000H -  007FH    0080H (128) bytes
 CODE:   0800H -  13D9H    0BDAH (3034) bytes
```

4.4.34 -q: Quiet Mode

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

4.4.35 -RAM*ranges*: Define RAM Ranges

The area of memory to be used for **far** data can be specified with the -A option, but this allows only for a contiguous area to be used from a base address to the end of memory. If there are multiple blocks of non-contiguous RAM available for **far** data, either due to physical memory layout or other requirements, the -RAM option allows these to be specified. Following the option are a comma-separated list of address ranges, in hex. For example

```
-RAM8000-ffff,20000-4ffff,1f000-1ffff
```

Each range should specify the lowest address to be used and the highest address to be used. The lowest address must be even, and the highest address must be odd. Make sure any ranges do not overlap the memory to be used for stack and other data allocated in the page zero segment.

4.4.36 -ROM*ranges*: Define ROM Ranges

Program code can be allocated into specific areas of the code memory space - the interrupt vectors and some other code must be allocated at a fixed address, and the ROM address specified in the -A option is always used for this. If a -ROM option is used, then rather than all other code being allocated upwards from above the vectors, it will be allocated into the specified ranges. The syntax is identical to the -RAM option, e.g.

```
-ROM0-FFFF,20000-37FFF
```

Note that it is not necessary to exclude areas used by the vectors from the ROM ranges, as the linker will do so automatically.

4.4.37 -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:

```
XAC -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.

4.4.38 -SA: Compile to Avocet assembler source files

This option directs XAC to compile the source to *Avocet* AVMAC assembler source files.

4.4.39 -STRICT: Strict ANSI Conformance

The -STRICT option is used to enable strict ANSI conformance of all special keywords. HI-TECH C supports the special keywords, such as **near**, **far** and **interrupt** which are used to allow access to the direct addressing mode under programmer control, and to handle interrupts using C code. If the -STRICT option is used, these keywords are changed to **__near**, **__far** and **__interrupt** respectively so as to strictly conform to the ANSI standard.

4.4.40 -TEK: Generate Tektronix HEX File

The -TEK option tells XAC to generate a *Tektronix* format HEX file if producing a file with .hex extension. This option has no effect if used with an option which specifies a .bin file output.

4.4.41 -U*macro*: Undefine a Macro

-U, the inverse of the -D option, is used to undefine predefined macros. For example, to remove the predefined macro **_XA_** use the option -U_XA_.This command, for example, is equivalent to:

```
#undef  _XA_
```

placed at the top of each module compiled using this option.

4.4.42 -UBROF: Generate UBROF Format Output File

The -UBROF option tells XAC to generate an 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 an option which specifies a .bin file output.

4.4.43 -UNSIGNED: Make *char* Type Unsigned

`-UNSIGNED` will make the default character type **`unsigned char`**. The default behaviour of `XAC` is to make all character values and variables **`signed char`** unless explicitly declared or cast to **`unsigned char`**. If `-UNSIGNED` is used, the default character type becomes **`unsigned char`** and variables will need to be explicitly declared **`signed char`**. The range of **`signed char`** is -128 to +127 and the range of **`unsigned char`** is 0 to 255.

4.4.44 -V: Verbose Compile

`-V` places the compiler in a verbose mode. 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.

4.4.45 -W*level*: 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. For example:

```
XAC -W-2 -C foo.c
```

4.4.46 -X: Strip Local Symbols

The `XAC` 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.

4.4.47 -Zg*[level]*: Global Optimisation

The `-Zg` option invokes global optimization during the code generation pass. This can result in significant reductions to code size and internal RAM usage.

The default level for this option is 1 (the least optimization). The level can be set anywhere from 1 to 9 (the most optimization).

Features and Run-time Environment

XA 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 specific to the *Philips* XA family of processors.

5.1 ANSI Standard Issues

5.1.1 Implementation-defined behaviour

Certain sections of the ANSI standard have implementation-defined behaviour. This means that the exact behaviour of some C code can vary from compiler to compiler. Throughout this manual are sections describing how the XA C compiler behaves in such situations.

5.2 Processor-related Features

XA C has many features which relate directly to the XA family of processors. These are detailed in the following sections.

5.2.1 Code Size Limitations of Functions and Modules

In the XA, functions have a code size limit of 64k. As modules are not divisible, they also have a code size limit of 64k. The sole reason for this limit is that an indirect jump (**`jmp [Rs]`**) only affects the low 16 bits of the program counter.

5.2.2 XA Hardware Design

There is one important consideration when designing an XA based system that is to efficiently run C code; due to the fact that in system mode the XA stack pointer is only 16 bits wide, with the upper part of the memory address forced to zero, the stack can only be located in the first 64k of data memory. Although the user mode stack pointer is 24 bits wide, there are a number of limitations on user mode that make it difficult to use for typical embedded applications.

Since the stack can only be located in the first 64k of memory, and other variables must be addressable in the same 64k segment as the stack (since the C language model does not differentiate between pointers to stack variables and other variables) this implies that DS must remain set to zero during program execution. This then means that external data memory must be mapped into the first 64k of the data address space.

Although in some circumstances it may seem advantageous to map the data memory at a higher address, the difficulties caused by this are considerable. Memory beyond 64k can be accessed using **`far`**

variables and pointers, but the compiler will manage this using ES. DS must remain at zero, and any attempt to configure a system in which DS is set to a non-zero value is unsupported by the compiler.

To implement a system where programs are to be downloaded into RAM or flash ROM for execution, the recommended approach is to map ROM and RAM at zero, as discussed above, but to also map the RAM at a higher address into code space, thus allowing programs to be executed out of RAM, or to map flash ROM at a higher address in the data space, so that it may be written to.

5.3 Files

5.3.1 Source Files

The extension used with source files is important as it is used by the compiler drivers to determine their content. Source files containing C code should have the extension `.c`, assembler files should have extensions of `.as`, relocatable object files require the `.obj` extension, and library files should be named with a `.lib` extension. See the tutorial Section 2.1.2 on page 18 for more information on how these input files are processed by the compiler.

5.3.2 Output File Formats

`XA C` 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 `HPDXA` integrated environment compiler driver you can select *Motorola* HEX, *Intel* HEX, binary, UBROF, *Tektronix* HEX, *American Automation* symbolic HEX, *Intel* OMF-51 or *Bytecraft* COD using the **Output file type...** menu item in the **Options** menu.

The default behaviour of the `XAC` command is to produce *Intel* HEX output. If no output filename or type is specified, `XAC` will produce an *Intel* HEX file with the same base name as the first source or object file. Table 5 - 1 on page 113 shows the output format options available with `XAC`. 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 XAC. 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 filename with a `.bin` type, for example `-Otest.bin`, `XAC` 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`.

5.3.3 Symbol Files

The `XAC -G` and `-H` options tell the compiler to produce a symbol file which can be used by debuggers and simulators to perform symbolic and source-level debugging. The `-H` option produces symbol files which contain only assembler level information whereas the `-G` option also includes C source level information. If no symbol filename is specified, by default a file called *`file`*`.sym` will be produced,

Table 5 - 1 Output File Formats

Format Name	Description	XAC Option	File Type
Motorola HEX	S1/S9 type hex file	-MOTOROLA	.hex
Intel HEX	*Intel* style hex records (default)		.hex
Binary	Simple binary image	-BIN	.bin
UBROF	Universal Binary Image Relocatable Format	-UBROF	.ubr
Tektronix HEX	*Tektronix* style hex records	-TEK	.hex
American Automation HEX	HEX format with symbols for *American Automation* emulators	-AAHEX	.hex
OMF-51	*Intel* Absolute Object Module Format	-OMF	.omf
Bytecraft .COD	*Bytecraft* code format	n/a	.cod

where *file* is the basename of the first source file on the command line. For example, to produce a symbol file called test.sym which includes C source level information:

```
XAC -Gtest.sym test.c
```

The symbol files produced by these options may be used with in-circuit emulators for the XA, for example those from *Nohau* and *Ashling*.

5.3.4 Avocet Symbol Tables

The XAC option -AV can be used in conjunction with the -H option to generate *Avocet* style symbol tables for use with the AVSIM simulator and certain in-circuit emulators. -AV should not be used with the -G option as the *Avocet* symbol table format does not support source-level debugging information.

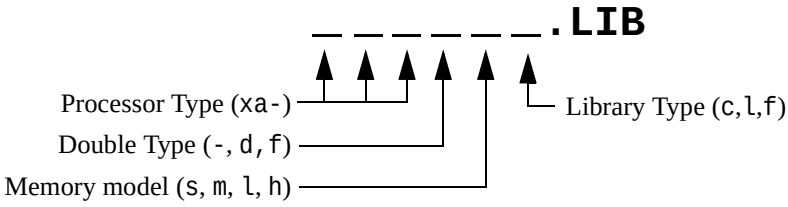
5.3.5 Standard Libraries

XA C includes a number of standard libraries, each with the range of functions described in the Library Functions chapter.

Figure 5 - 1 on page 114 illustrates the naming convention used for the standard libraries. The meaning of each field is described here, where:

- *Processor Type* is always xa-.
- *Double Type* is - for 32-bit doubles, d for 64-bit doubles and f for fast 64-bit doubles.
- *Memory model* is s for small, m for medium, l for large and h for huge.
- *Library Type* is c for standard library, l for the library which contains only printf-related functions with additional support for longs, and f for the library which contains only printf-related functions with additional support for longs and floats.

Figure 5 - 1 XA Standard Library Naming Convention



5.3.6 Run-time Startup Modules

A C program requires certain objects to be initialised and the processor to be in a particular state before it can begin execution. It is the job of the *run-time startup* code to ready the program for execution. Since this is code that executed before the C program, it is necessarily written in assembler code. The run-time startup code is executed almost immediately after reset. In fact it is called by a special *powerup* routine, described below, that is directly located at the reset vector address. For the XA processors, the principle job of the run-time startup code is to initialise the stack pointer, select the XA memory model, clear uninitialized variables and assign values to those variables that have been initialised.

5

5.3.6.1 Stack pointer and XA memory model

The run-time startup code first places the XA processor into one of two memory modes. Note the distinction between the memory model of the processor and the memory model used by the compiler. The two XA processor memory models affect the data pushed onto the stack when subroutines are called. The compiler must be aware of how many bytes of address are present on the stack after a call so that it may retrieve function parameters that were pushed onto the stack before the call was made. For this reason you should not attempt to alter the default XA memory model selected by the run-time code associated with each memory model.

The compiler's small and medium memory models set the PZ bit in the system configuration register (SCR). This places the XA processor into Page 0 memory mode. In this mode, the XA chip maintains 16 address bits so all memory is limited to 64 k bytes. Only a 16-bit address is pushed onto the stack during subroutine calls and interrupts.

In large and huge compiler memory models, the XA processor is placed into its large memory model which allows full 24 bit addressing and access to the full address space. The full 24-bit address is pushed onto the stack during calls and interrupts.

In all compiler memory models, the run-time startup code initialises the system stack pointer to be the address specified for the `stack` psect in the linker options. This address can be specified using the `HPDXA` **ROM & RAM addresses...** dialog or the `XAC -A` command-line option.

5.3.6.2 Clearing the bss, rbss and rbit psects

The run-time startup code will clear, or assign the value zero, any variables which are uninitialized at their definition and which are non-auto. This amounts to those objects which have been placed in the `bss`, `rbss` or `rbit` psects. Since these psects are defined as a contiguous block of memory, the run-time startup code calls a routine to clear a block of memory for each psect. In the following example, all but the object **`loc`** will be cleared by the startup code since it is an **`auto`** object. The initial value of **`loc`** is unknown.

```
int          i;
near int     ni;
bit          b;
void main(void)
{
       static int   sloc;
       int          loc;
...
```

The other function of the run-time startup code is to initialise those variables assigned a value at their definition. This amounts to a block copy of the initial values from ROM to the RAM areas designated for those objects. Code to perform the copy is only included if required. In this example:

```
int          i  = 7;
near int     ni = 6;
bit          b;
void main(void)
{
       static int   sloc = 5;
       int          loc  = 7;
...
```

The objects **`i`**, **`ni`** and **`sloc`** will be initialised by the run-time startup code. Note that you cannot initialise **`bit`** objects and that initialised automatic variables are assigned their starting value by code placed within the function in which they are defined.

Any objects defined in assembler code, i.e. they have memory reserved using any of the **`DS`**, **`DB`**, **`DF`**, **`DD`** or **`DW`** assembler directives, will also be cleared or initialised at startup providing that the directives are placed within the compiler-generated psect used for C variables, such as `bss`, `rbss`, `data` or `rbit` etc.

The run-time startup code calls the function **`main()`**, which is referred to as **`_main`** by the run-time startup code. Note the *underscore* "_" prepended to the function name. The function **`main()`** is, by definition of the C language, the "main program".

The compiler is supplied with run-time startup modules for each of the memory models and processors which are supported. The standard run-time startup modules, found in the LIB directory, are listed in Table 5 - 2 on page 116.

Table 5 - 2 Standard Run-time Startup Modules

Filename	Use
rtxa--s.obj	Small model startup module
rtxa--m.obj	Medium model startup module
rtxa--l.obj	Large model startup module
rtxa--h.obj	Huge model startup module

The source code used to generate the run-time startup module is called `rtxa--`*`m`*`.as` , where *m* is the memory model key, which are in the SOURCES directory of your distribution.

5.3.6.3 Memory models and the data Psect

In the small model, the `data` psect is linked directly into ROM with the same link and load addresses. `Data` is not modifiable in the small model, and is accessed directly using **`movc`** instructions. In the medium and large models, the `data` psect is linked with a load address in ROM and a link address in external RAM. When the medium or large model run-time startup code is executed, it copies the `data` psect from ROM to RAM.

5.3.7 Customising the Run-time Startup Code

If you find that you are running out of ROM space, you may wish to reduce the ROM size further by customising the run-time startup code. The standard run-time startup modules contains code to clear the `rbit`, `rbss`, and `bss` psects and copy the `rdata` psect. The medium and large model run-time startup modules also contain code to copy the `data` psect. Code space be saved by creating a custom version of the run-time startup code which performs only those initialisations required by your application. The source code for the run-time startup code may be found in the SOURCES subdirectory, if installed. Several different versions of the run-time startup code are supplied, these are listed in Table 5 - 2 on page 116.

5.3.7.1 Copyright Notice

The run-time startup code includes a HI-TECH C Software copyright notice which is linked at the start of ROM. If you are running out of code space you may wish to delete the copyright notice, saving a small amount of ROM. If ROM space permits, the HI-TECH Software copyright notice should be left intact. You may also wish to add your own copyright notice.

5.3.8 Using the New Run-time Module Code

Once you have modified the run-time code to suit your needs, reassemble it using the command:

```
asxa -x rtxa--m.as
```

where `rtxa--m.as` is the name of the run-time module which you have modified. The run-time code should be copied to the LIB directory. This is the subdirectory identified by the HTC_XA_LIB environment variable, for most installations it will be `c:\ht-xa\lib`.

Once the modified run-time code is present in the library directory, you need only relink your application to use your modifications. If you are using customised run-time code, you need to be careful when making further modifications to your application. If you are running a minimal run-time module and add code which uses the `bss` psect or RAM based `data` you may have problems. Without the `bss` clear code, any uninitialised variables in external RAM will not be cleared to zero before **`main()`** is invoked. More importantly, if the initialised data is used, but the `data` or `rdata` copy code is missing, statically initialised variables will not be set to their correct values at startup. This problem may also effect **`bit`** variables if you remove the code which clears the **`rbit`** psect.

5.3.9 The powerup Routine

Some hardware configurations require special initialisation, often within the first few cycles of execution after reset. The watchdog timer in the XA is an example of this - to disable it, it must be programmed off immediately after reset. Rather than having to modify the run-time 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, e.g. this code turns off the watchdog timer:

```
#asm

        psect   text,global,reloc=2,align=2
        global  powerup,start
powerup:
        clr     0x02fa          ;clear watchdog enable flag
        mov.b   0x45d,#0xa5     ;watchdog1
        mov.b   0x45e,#0x5a     ;watchdog2
        jmp     start
#endasm
```

Note that the code ends with a jump to **`start`**, which is the entry point in the run-time module. The code in the powerup routine can set up RAM, chip selects etc., since no stack, RAM or any other resources are assumed when it is executed.

5.4 Supported Data Types and Variables

The XA compiler supports basic data types of 1-bit, 8-bit, 16-bit and 32-bit 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. The XA has a requirement that word accesses be on word boundaries, so that all objects 2 bytes in size or larger will be allocated on an even address. Unions are aligned to the word boundary. Structures will be padded to an even number of bytes in size, and word-sized or larger structure members will be aligned to even offsets. Thus structures may contain "holes".

Table 5 - 3 shows the data types and their corresponding size and arithmetic type.

Table 5 - 3 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	32	real
double	32 or 64[b]	real

a.A char is signed by default, and unsigned if the XAC `-UNSIGNED` option is used.
b.A double defaults to IEEE 32-bit, but becomes IEEE 64-bit with the XAC `-DOUBLE` option, or fast double 64-bit format with the XAC `-FDOUBLE` option.

5.4.1 Radix Specifiers and Constants

The format of integral constants specifies their radix. XA C 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 - 4 on page 119. The letters used to specify binary or hexadecimal radices are case insensitive, as are the letters used to specify the hexadecimal digits.

Any integral constant will have a type which is the smallest type that can hold the value without overflow. The suffix **`l`** or **`L`** may be used with the constant to indicate that it must be assigned either a **`signed long`** or **`unsigned long`** type, and the suffix **`u`** or **`U`** may be used with the constant to indicate

Table 5 - 4 Radix Formats

Radix	Format	Example
binary	0b*number* or 0B*number*	0b10011010
octal	0*number*	0763
decimal	*number*	129
hexadecimal	0x*number* or 0X*number*	0x2F

that it must be assigned an unsigned type, and both **l** or **L** and **u** or **U** may be used to indicate **unsigned long int** type.

Floating-point constants have **double** type unless suffixed by **f** or **F**, in which case it is a **float** constant. The suffixes **l** or **L** specify a **long double** type which is identical to **double** with XA C.

Character constants are enclosed by single quote characters " **'** ", for example **'a'**. A character constant has char type. Multi-byte character constants are not supported.

String constants or string literals are enclosed by double quote characters " **"** ", for example **"hello world"**. The type of string constants is **const char *** and the strings are stored in ROM.

XA C will use the same storage location and label for strings that have identical character sequences, except where the strings are used to initialise an array residing in RAM.

Two adjacent string constants (i.e. two strings separated *only* by white space) are concatenated by the compiler. Thus:

```
const char * cp = "hello " "world";
```

assigned the pointer with the string **"hello world"**.

5.4.2 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 psect rbit, 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.

The rbit psect is linked into the XA bit-addressable area from 20h to 3Fh, limiting the number of **bit** variables in a single program to 256.

Note that when assigning a larger integral type to a **bit** variable, only the least-significant bit is used. For example, if the **bit** variable **bitvar** was assigned as in the following:

```
int    data = 0x54;
bit    bitvar;

bitvar = data;
```

it will be cleared by the code since the least significant bit of **data** is zero.

If you want to set a **bit** variable to be 0 or 1 depending on whether the other value is zero (false) or non-zero (true), use the form:

```
bitvar = data != 0;
```

The psects in which **bit** objects are allocated storage 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 units of bits, not bytes. All addresses specified for bit objects are also bit addresses.

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

Bit variables are manipulated using the efficient XA bit-addressing modes. These 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. Due to the absence of suitable addressing modes on the XA it is not possible to declared pointers to **bit** variables or statically initialise **bit** variables. If the XAC flag -STRICT is used, the **bit** keyword becomes __**bit**.

5.4.2.1 Using Bit Addressable SFRs

The **bit** variable facility may be combined with absolute variable declarations to access the bit-addressable special function registers (SFRs) at bit addresses 200h to 3FFh. The 512 bit addresses from 200h to 3FFh map onto the 64 special function registers with addresses from 400h to 43Fh.

For example, to access bit 3 of port P2 at 432h, declare a **bit** variable at absolute address 393h:

```
static bit         P2_3 @ 0x393;
```

Similarly, bits 0 to 7 of port P0 at address 430h would be declared as:

```
static bit         P0_0 @ 0x380;
static bit         P0_1 @ 0x381;
static bit         P0_2 @ 0x382;
```

```
static bit          P0_3 @ 0x383;
static bit          P0_4 @ 0x384;
static bit          P0_5 @ 0x385;
static bit          P0_6 @ 0x386;
static bit          P0_7 @ 0x387;
```

5.4.3 8-Bit Integer Data Types

HI-TECH C supports both **signed char** and **unsigned char** 8-bit integral types. The default **char** type is **signed char** unless the XAC option -UNSIGNED is used, in which case the default is **unsigned 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**, **int** and **long int** in C expressions.

On the XA 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. **Unsigned char** is the C type which logically maps onto the format of most XA special function registers. The **unsigned char** type is the most efficient data type on the XA and maps directly onto the 8-bit bytes which are most efficiently manipulated by XA instructions. It is suggested that **char** types be used wherever possible so as to maximise performance and minimise code size.

Variables may be declared using the **signed char** and **unsigned char** keywords, respectively, to hold values of these types. Where only **char** is used in the declaration, the type will be **signed char** unless the option, mentioned above, to specify un**signed char** as default is used.

5.4.4 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.

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. 16-bit integers 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.

Variables may be declared using the **signed int**, **unsigned int**, **signed short int** and **unsigned short int** keyword sequences, respectively, to hold values of these types. Where only **int** is used in the declaration, the type will be **signed int**. When specifying a **short int** type, the keyword **int** may be omitted. Thus a variable declared as **short** will contain a **signed short int** and a variable declared as **unsigned short** will contain an **unsigned short int**.

5.4.5 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. 32-bit integer values are represented in *little endian* format with the least significant word and least significant byte at the lowest address. 32 bits are used for **long** and **unsigned long** as this is the smallest long integer size allowed by the ANSI standard for C.

Variables may be declared using the **signed long int** and **unsigned long int** keyword sequences, respectively, to hold values of these types. Where only **long int** is used in the declaration, the type will be **signed long**. When specifying this type, the keyword **int** may be omitted. Thus a variable declared as **long** will contain a **signed long int** and a variable declared as **unsigned long** will contain an **unsigned long int**.

5.4.6 Floating-point

Floating-point is implemented using IEEE 32-bit, IEEE 64-bit, or fast double (64-bit) format. By default the **float** and **double** types are IEEE 32-bit values, formatted as follows:

Table 5 - 5 IEEE 32-bit Floating-point Format

31	30 23	22 0
Sign - 1 bit	Exponent - 8 bits	Mantissa - 23 bits plus one hidden bit

The byte containing the sign and seven of the eight exponent bits is at the highest address. If the XAC option -DOUBLE is used, or double precision floating-point is selected in HPDXA, then **double** becomes 64-bit IEEE format, as follows:

Table 5 - 6 IEEE 64 bit Floating-point Format

63	62 52	51 0
Sign - 1 bit	Exponent-11 bits	Mantissa - 52 bits plus one hidden bit

Note that when using 64-bit floating-point, **float** variables are still 32-bit, but functions like **printf()** that expect a floating-point argument will convert any 32-bit float value to 64 bits. If the XAC option -FDOUBLE is used, or fast double format is selected in HPDXA, then **double** becomes 64-bit fast double precision floating-point format, as follows:

Table 5 - 7 64 bit Fast Double Format

63	62 48	47 0
Sign - 1 bit	Exponent-15 bits	Mantissa - 48 bits

Note that when using 64-bit fast double format, **float** variables are still in IEEE 32-bit format. Using the fast double format, the arithmetic operations are substantially faster than using IEEE floating-point format.

5.4.7 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. Structure members larger than one byte will be aligned on an even boundary, and structure sizes will be padded to an even value if required.

5.4.8 Bit Fields in Structures

HI-TECH C fully supports *bit fields* in structures. Bit fields are allocated starting with the least significant bit. Bit fields are allocated within 16-bit words, the first bit allocated will be the least significant bit of the least significant byte of the word. Bit fields are always allocated in 16-bit units, starting from the most significant bit. When a bit field is declared, it is allocated within the current 16-bit unit if it will fit, otherwise a new 16-bit word is allocated within the structure. Bit fields never cross the boundary between 16-bit words, but may span the byte boundary within a given 16-bit allocation unit. For example, the declaration:

```
struct {
        unsigned        hi : 1;
        unsigned        dummy : 14;
        unsigned        lo : 1;
} foo @ 0x10;
```

will produce a structure occupying 2 bytes from address 10h. The field **hi** will be bit 0 of address 10h, **lo** will be bit 7 of address 11h. 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 11h. If a bitfield is declared in a structure that is assigned an absolute address, no storage will be allocated, and the fact that 16 bits are reserved is unimportant, so to model a byte location with bit fields, you may simply define the bits as though only one byte was occupied.

Unnamed bitfields 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           : 14;
        unsigned        lo : 1;
} foo @ 0x10;
```

A structure with bitfields may be initialised by supplying a comma-separated list of initial values for each field. For example:

```
struct {
        unsigned        hi  : 1;
        unsigned        mid : 6;
        unsigned        lo  : 1;
} foo = {1, 8, 0};
```

5.4.8.1 Structure and Union Qualifiers

XA C supports the use of type qualifiers on structures. When a qualifier is applied to a structure, all of its members will inherit this qualification. In the following example the structure is qualified **`const`**.

```
const struct {
       int number;
       int *ptr;
} record = { 0x55, &i};
```

In this case, the structure will be placed into ROM and each member will, obviously, be read-only. Remember that all members must be initialised if a structure is **`const`**.

If the members of the structure were individually qualified **`const`** but the structure was not, then the structure would be positioned into RAM, but each member would be read-only. Compare the following structure with the above.

```
struct {
       const int number;
       int * const ptr;
} record = { 0x55, &i};
```

5.4.9 Standard Type Qualifiers

Type qualifiers provide information regarding how an object may be used, in addition to its type which defines it storage size and format. XA C supports both ANSI qualifiers and additional special qualifiers which are useful for embedded aplications and which take advantage of XA architecture.

5.4.9.1 Const and Volatile Type Qualifiers

HI-TECH C supports the use of the ANSI keywords **const** and **volatile**. The **const** keyword 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 psect called const. For example:

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

The **volatile** keyword 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 I/O ports and any variables which may be modified by interrupt routines should be declared **volatile**, for example:

```
volatile unsigned char  P1 @ 0x430;
```


5.4.10 Special Type Qualifiers

XA C supports special keywords, **near**, **far**, **huge**, **persistent** and **code** to allow the user to control placement of **static** and **extern** class variables into particular address spaces. If the XAC -STRICT option is used, these keywords are changed to **__near**, **__far**, **__huge**, **__persistent** and **__code**. These keywords may also be applied to pointers. These keywords 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. You may not write:

```
void func(void)
{
  near int intvar;  /* Wrong! */
  .. other code ..
}
```

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

```
static near int intvar;
```

5.4.10.1 *Near* Qualifier

The **near** keyword is used to place variables in internal RAM, where they may be more efficiently manipulated using XA direct addressing. This keyword is of most use in the medium and large models which access **static** variables by indirection through registers. In the small model all **static** and **extern** variables are placed in internal RAM so the **near** keyword need not be used. Variables declared to be **near** are placed in the psect rbss, which is linked into internal RAM in all memory models. Use of **near** can provide substantial improvements to code quality.

Near variables may be statically initialised, for example:

```
static near int  bufsize = 128;
```

Initialised **near** variables such as **bufsize** will be placed in a psect called rdata and copied from ROM to internal RAM by the run-time startup module.

5.4.10.2 *Far* Qualifier

The keyword **far** is used to place objects in banked RAM (that is, outside the RAM bank currently addressed by the DS register). The **far** keyword is used to declare **static** variables as follows:


```
far int  f_int;
```

All accesses to **f_int** will use indirection via the ES register to access the bank in which the variable is located.

Pointers to objects of class **far** may be declared:

```
far char * fptr;
```

A **far** pointer can access variables in the current bank or any other bank. If you take the address of a variable in the current bank and assign it to a **far** pointer, the contents of DS will be used to give the upper 8 bits of the pointer. Note that **far** pointers occupy 32 bits even though only 24 bits are used.

5.4.10.3 *Persistent* 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** keyword 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, and this area of memory may be assigned to a specific address (with the -A option to XAC, or in the HPDXA **ROM and RAM addresses...** menu under **Options**). Thus if a small amount of non-volatile RAM is provided then **persistent** variables may be assigned to that memory. On the other hand if all memory is non-volatile, you may choose to have **persistent** variables allocated to addresses by the compiler along with other variables (but they will still not be cleared). One advantage of assigning an explicit address for

persistent variables is that this can remain fixed even if you change the program, and other variables get allocated to different addresses. This would allow configuration information etc. to be preserved across a firmware upgrade.

There are some library routines provided to check and initialise **persistent** data - see page 322 for more information, and for an example of using **persistent** data.

If the **persistent** keyword is combined with the **far** keyword, then the variable will be allocated in another psect (farnvram) which can be located anywhere in the full address space of the XA. This is useful if you have non-volatile RAM mapped into a high address location. Note that the functions described on page 322 do *not* handle **persistent far** data. However since the source code for these routines is supplied, it would be possible to modify these to work with **far** data.

5.4.11 *Code* Qualifier

The **code** keyword is used to place initialised static objects into the CODE address space of the XA. Objects declared to be **code** must be statically initialised, for example:

```
code int count = 0x1234;
```

The compiler will place **count** in the code psect, which is linked into program ROM immediately after the text psect. **Code** objects can not be modified, as the XA architecture has no instructions defined that will write to code memory.

All access to **code** objects takes place via the **movc** instruction. Objects of class **code** occupy a completely separate address space to normal variables and constants. Standard pointers and **far** pointers cannot even address objects of class **code** in the medium and large models. In small model it is not necessary to use the **code** qualifier as any initialised data (other than **near** data) is automatically placed into ROM.

In order to access data of class **code**, the XA compiler supports pointers to **code**. The most common application of **code** objects is to store strings in ROM. Clearly, such strings cannot be accessed by routines like **printf()** and **puts()** which accept normal **char *** arguments. A routine to write a **code** string in the same manner as **puts()** could be encoded as:

```
void code_puts(code char * codeptr)
{
        char    ch;
        while (ch = *codeptr++)
                putch(ch);
        putch('\n');
}
```

code_puts() could then be used to display strings directly from ROM as follows:

```
extern void     code_puts(code char *);
code char       hello[] = "Hello, world\n";
main()
{
        code_puts(hello);
}
```

Care must be taken to avoid passing pointers to **code** to any routine which expects a default or **far** pointer. Likewise, normal and **far** pointers should not be passed to routines like **code_puts()**. Naturally the compiler will assist with appropriate warning messages if it detects conversions between incompatible pointer classes.

When using the large model, **code** class data may be placed anywhere in the code space. This means that pointers to **code** will be 32-bit but only 16-bit in small and medium model.

5.4.12 Pointer Types

HI-TECH C supports several different classes of pointer, of both 16- and 32-bit size. 16-bit pointers may only access objects that reside in the current RAM bank. 32-bit pointers may be used to access objects in the current RAM bank, extended RAM banks and the CODE address space depending on the usage and the class of the pointer.

The default pointer class, unmodifed by any class keywords such as *code*, is a 16-bit pointer which addresses a 64k address space which is the concatenation of internal RAM (00h to FFh) and either the CODE space (in small model), or the current RAM bank space (in the medium and large models).

The following tables gives the pointer sizes for different pointer types in the available memory models. A “pointer to near” is any pointer which has **near** on the left side of the ***** in its declaration, for example: **near int * nip;** or **float near * nfp;** A default pointer has no qualifier on the left side of the *****.

Table 5 - 8 XAC Pointer sizes (in bytes) and memory models

pointer types	small	medium	large	huge
default pointer	2	2	2	4
pointer to near	n/r	2	2	2
pointer to far	n/a	2	4	4
pointer to code	n/r	2	4	4
pointer to huge	n/a	n/a	4	n/r
function pointer	2	2	4	4

Some pointer types are illegal in certain memory models. These are indicated with “n/a” in the table. Some pointer types are not required to be used in certain memory models, since the default type pointer can be used. These are indicated with “n/r” in the table.

It is important to remember that in small model, pointers to addresses above the highest allocated RAM address do not address any RAM.

5.4.12.1 Combining Type Qualifiers and Pointers

Pointers can be qualified like any other C object, but care must be taken when doing so as there is are two quantities associated with pointers. The first is the actual *pointer* itself, which is treated like any ordinary C variable and has memory reserved for it. The second is the *object* that the pointer references, or to which the pointer points. The general form of a pointer definition looks like the following.

“object’s type & qualifiers“ * “pointer’s qualifiers” “pointer’s name” ;

Here are three examples, highlighting the fields with spacing:

```
near int          *                        nip ;
int               *        near            inp ;
near int          *        near            ninp;
```

The first example is a pointer called **nip**. It contains the address of an **int** object that is qualified **near**. The object will be located in internal RAM and directly accessed. The pointer itself is unqualified and so will reside at a location dictated by the memory model.

The second example is a pointer called **inp** which contains the address of an **int** object. Since this object is not **near** and so will reside at a location dictated by the memory model. The **near** keyword after the ***** indicates that the pointer itself has been qualified **near** and so the pointer will reside in internal memory and be accessed using the direct addressing mode.

The last example is of a pointer called **ninp** which is itself qualified **near** and which also contains the address of an object that is also **near**. In this example, both the pointer and the object that the pointer references will be located in internal memory and directly accessed.

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.

The **const**, **volatile**, **persistent**, **far**, and **code** modifiers may also be applied to pointers in the above manner. Some of these pointer types are discussed below. XA C also employs the use of the qualifier **huge** which may only be applied to pointer types. This is also discussed below.

5.4.13 *Code* Pointers

The **code** keyword is used to declare constants which are placed in ROM and accessed using the **movc** instruction. HI-TECH C allows variables of class pointer to code to be declared. Pointers to code are of most use in the medium and large models where the default pointer class addresses RAM. Small model code will not need to use the **code** qualifier. A common use of code constants and variables of class pointer to code is to access string constants such as menus and prompts which have been placed in ROM.

The following code illustrates this technique:

```
#include        <conio.h>
static code char        hello[] = "Hello, world\n";
static void
code_puts(code char * cptr)
{
        char    ch;
        while (ch = *cptr++)
                putch(ch);
}
main()
{
        code_puts(hello);
}
```

Use of **code** constants and pointers can reduce external RAM usage, particularly in the medium and large memory models, which copy initialised variables to external RAM.

5.4.14 *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 small model, **const** pointers always access program ROM because objects declared **const** are stored in ROM. In the medium and large models const pointers behave like normal pointers except that you may not write to memory via a **const** pointer. The **const** class may be combined with the classes **near** and **far** to produce variables and pointers to read-only objects in either of these address spaces. Thus the declaration:

```
far const * ncptr;
```

produces a variable **ncptr** which is a 32-bit pointer to **const** characters in banked RAM. It is possible to read RAM by dereferencing **ncptr**, but the statement:

```
*ncptr = 0;
```

will be rejected by the compiler since you cannot write to objects qualified **const**.

5.4.14.1 *Huge* Pointer Qualifier

In large model, the **huge** type qualifier can be applied to an object pointed to by a pointer, to allow the pointer to reference either code or data spaces. A pointer to **huge** is always 32 bits, representing 24 bits of memory address, and a flag that indicates code or data. The flag is the most significant bit of the pointer, and is set for a code reference, and cleared for a data reference. The **huge** qualifier may also be used with huge model, but has no effect, since all pointers are **huge** in huge model. The **huge** qualifier may not be used in medium or small model, since no memory outside the lower 64k is used.

5.5 Storage Class and Object placement

Objects are positioned in different memory areas dependant on their storage class and declaration. This is discussed in the following sections.

5.5.1 Local Variables

A *local variable* is one which only has scope within the block in which it was defined. That is, it may only be referenced within that block. C supports two classes of local variables in functions: **auto** variables which are normally allocated on some sort of stack, and **static** variables which are always given a fixed memory location and have permanent duration.

5.5.1.1 *Auto* Variables

Auto (short for *automatic* since they automatically “appear” and “disappear” with block entry and exit) 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 on the stack and referenced by indexing off the stack pointer (R7). The variables will not necessarily be allocated in the order declared - in contrast to parameters which are always in lexical order. The usual alignment considerations apply, that is, any word or larger variables will be placed on an even address. The compiler sorts variables by size to

minimise holes left for alignment purposes. 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**.

5.5.1.2 *Static* Variables

Static variables are allocated in the bss psect, or the rbss psect if the object is also of type **near** and occupy fixed memory locations which will not be overlapped by storage for other functions. **Static** variables are local in scope to the function in which they are declared, but may be accessed by other function 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 XA.

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.

5.5.2 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 P1 @ 0x430;
```

will declare a variable called **P1** located at 430h (which is in the special function register area of the XA address space). Note that the compiler does not reserve any storage, for this object but merely equates the variable to that address. The compiler-generated assembler will include a line of the form:

```
_P1  equ  430h
```

Absolute variables provide a convenient method of accessing the built in special function registers of the XA. 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. 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 152.

If a variable is to be defined at an address above 64k, then it must also be qualified **far**, e.g.

```
far unsigned char a_port @ 0x20000;
```

5.6 Functions

5.6.1 The *noregsave* Function Qualifier

The **noregsave** special keyword is available for use with functions that never return, or where the function is the main program of a task under a real-time O/S. In either case, no general registers are saved.

If the XAC -STRICT option is used, the **noregsave** keyword is changed to **__noregsave**.

5.6.2 Function Argument Passing

The method used to pass function arguments depends on the size of the argument or arguments.

As described above, the first three arguments to a function are passed in registers if they are 16 bits in size or less. The registers used are R3, R2 and R1 (R3L, R2L and R1L for 8-bit values). Any arguments larger than 16 bits will be passed on the stack, as will any arguments after the first three. Note that if one of the first three arguments is passed on the stack, its corresponding register will *not* be used for any other argument.

Where a function has a variable argument list, that is, it has an *ellipsis symbol* "**...**" in its prototype, the last argument before the *ellipsis* will always be passed on the stack to provide an anchor point for the variable part of the argument list. The variable arguments themselves will be passed on the stack in order. Byte-size arguments on the stack will have an extra byte of padding, i.e. each parameter occupies a multiple of 16 bits on the stack.

If any stack parameters are passed, it is the responsibility of the calling function to adjust the stack pointer after the call returns.

5.6.3 Function Return Values

Function return values are passed to the calling function as indicated in the following sections.

5.6.3.1 8-Bit Return Values

8-bit values (**char** and **unsigned char**) are returned in register R0L. For example, the C function:

```
char
return_zero(void)
{
        return 0;
}
```

will exit with the following code:

```
        mov     R0L,#0
        ret
```

5.6.3.2 16-Bit Return Values

16-bit values (**int**, **short** and 16-bit pointer objects) are returned in R0.

```
int
test(void)
{
        return 0x1234;
}
```

will return with 0x1234 in R0.

5.6.3.3 32-Bit Return Values

32-bit values (**long**, **float** and 32-bit pointer objects) are returned in registers R0 and R1. The most significant word is in R1. This is illustrated by the following code:

```
long
return_long(void)
{
        return 0x01020304;
}
```

which will exit using the sequence of instructions:

```
        mov     R0,#304h
        mov     R1,#102h
        ret
```

5.6.3.4 64-Bit Return Values

Double float values, when the 64-bit double option is in effect, are returned in the registers R0, R1, R2 and R3, with the least significant bits in R0.

5.6.3.5 Structure Return Values

Composite return values (**struct** and **union**) are returned by various means depending on size. 8-bit structures are returned in register R0L, 16-bit structures in R0 and 32-bit structures in R0 and R1. Where a function returns a larger structure, the calling function supplies a parameter (in R0) which is a pointer to a structure to hold the return value. The called function saves this pointer, then uses it on exit to copy the return structure.

5.7 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. Certain classes of variable are assumed to reside within particular address ranges and address spaces, as limited by the XA architecture. For example any variable defined to lie at an absolute address in the range 400h-7FFh will be assumed to be an SFR and accessed with direct addressing only. The external data memory that shadows this address range will be accessible with indirect addressing like all other external data memory.

5.8 Register Usage

Compiled code always assumes that register bank 0 is selected. Registers R0, R1, R2 and R3 are used as temporary values and for indirectly addressing data in internal RAM. Registers R3, R2 and R1 are used for register-based argument passing and for function return values. These registers will also be used to hold temporary values within functions and may also be used to contain arguments or local variables if code is compiled with global optimisation. Registers R4, R5 and R6 are used to hold register variables. These registers should be preserved by any assembly language routines which are called.

5.9 Operators

XA C supports all the ANSI operators. The exact results of some of these are implementation defined. The following sections illustrate code produced by the compiler.

5.9.1 Integral Promotion

When there is more than one operand to an operator, they all must be of exactly the same type. The compiler will automatically convert the operands, if necessary, so they have same type. The conversion is to a “larger” type so there is no loss of information. Even if the operands have the same type, in some situations they are converted to a different type before the operation. This conversion is called *integral promotion*. XA C performs these integral promotions where required. If you are not aware that these changes of type have taken place, the results of some expressions are not what is expected.

Integral promotion is the implicit conversion of enumerated types, signed or unsigned varieties of **`char`**, **`short int`** or bitfield types to either **`signed int`** or **`unsigned int`**. If the result of the conversion can be represented by an **`signed int`**, then that is the destination type, otherwise the conversion is to **`unsigned int`**.

Consider the following example.

```
unsigned char count, a=0, b=50;
if(a - b < 10)
  count++;
```

Certainly the **unsigned char** result of **a - b** is 206, but both **a** and **b** are converted to **signed int** via integral promotion before the subtraction takes place. The result of the subtraction with these data types is -50 and hence the body of the **if()** statement is executed. If the result of the subtraction is to be an unsigned quantity, then apply a cast. For example:

```
if((unsigned int)(a - b) < 10)
  count++;
```

The comparison is then done using **unsigned int**, in this case, and the body of the **if()** would not be executed.

Another problem that frequenct occurs is with the bitwise compliment operator, "**~**". This operator toggles each bit within a value. Consider the following code.

```
unsigned char count, c;
c = 0x55;
if( ~c == 0xaa)
  count++;
```


If **c** contains the value 55h, it tempting to believe that **~c** would produce AAh, however the result is FFAAh and so the comparison in the above example would fail. The compiler may be able to issue a mismatched comparison error to this effect is some circumstances. Again, a cast could be used to change this behaviour.

The consequence of integral promotion as illustrated above is that operations are not performed with char type operands, but with int type operands. However there are circumstances when the result of an operation is identical regardless of whether the operands are of type char or int. In these cases, XA C will not perform the integral promtion so as to increase the code efficiency. Consider the following example.

```
unsigned char a, b, c;
a = b + c;
```

Strictly speaking, this statement requires that the values of **b** and **c** should be promoted to **unsigned int**, the addition performed, the result of the addition cast to the type of **a**, and then the assignment can take place. Even if the result of the **unsigned int** addtion of the promoted values of **b** and **c** was different to the result of the **unsigned char** addition of these values without promotion, after the **unsigned int** result was converted back to **unsigned char**, the final result would be the same. An 8-bit addition is more efficent than an a 16-bit addition and so the compiler will encode the former.

If, in the above example, the type of **a** was **unsigned int,** then integral promotion would have to be performed to comply with the ANSI standard.

5.9.2 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.

XA C 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.

5.9.3 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 - 9 shows the expected sign of the result of the division of operand 1 with operand 2 when compiled with XA C.

Table 5 - 9 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.

5.10 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 XAC or HPDXA 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, or write your own assembly language subroutines you should read this section carefully.

A psect can be created in assembler code by using the **PSECT** assembler directive, see. In C, user-defined psects can be created by using the **#pragma psect** directive.

5.10.1 Compiler-Generated Psects

The code generator places code and data into psects with standard names which are subsequent positioned by the default linker options for the memory model you are using. These psects are described below.

`vectors` The `vectors` psect contains the reset vector followed by all initialised interrupt vectors. `Vectors` is normally linked for address 0 in ROM so that the XA will fetch the reset vector contents from zero on reset.

`text` is used for all executable code. By default the C compiler places all executable code in the `text` psect. User written assembly language subroutines should also be placed in the `text` psect. When using the large model, C code will be placed in `ltext` psects, which will be distributed over various banks, but the `text` psect will always be in bank 0. For this reason all interrupt functions are placed in the `text` psect. In the small and medium models all code is placed in the `text` psect.

`code` is used for any statically initialised constants of class **`code`**. For example:

```
code int        maxdata = 10;
```

declares a constant **`maxdata`** with value 10 which resides in the `code` psect. `Code` is linked into program ROM after the `text` psect. Objects in the `code` psect are accessed using the **`movc`** instruction.

`const` is used for all initialised constants of class **`const`**, for example:

```
const char      masks[] = { 1,2,4,8,16,32,64,128 };
```

`strings` The `strings` psect is used for all unnamed string constants, such as string constants passed as arguments to routines like **`printf()`** and **`puts()`**.

`data` The `data` psect is used to contain all statically initialised data except those in classes **`near`**, **`code`** and **`const`**.

For the small memory model, the `data` psect is linked into ROM, statically initialised data items are not modifiable and are accessed using the **`movc`** instruction.

For the medium and large memory models, the `data` psect is linked into RAM, with a copy in ROM which is transferred to RAM by the run-time startup code. Statically initialised data items may be modified like any other variable in the medium and large models.

`rdata` contains all statically initialised variables of class **`near`**, for example:

```
static near int size = 256;
```

The rdata psect behaves in the same manner for all memory models. Initialised data will only be placed in rdata if declared to be **near**, otherwise it will be placed in the data psect.

A copy of the rdata psect is stored in ROM and transferred to internal RAM by the run-time startup code before **main()** is invoked.

bss — The bss psect is used for all uninitialised **static** and **extern** variables which reside in external RAM. Bss is cleared to all zeros by the run-time startup code before **main()** is invoked.

rbss — contains any uninitialised variables of class **near**. The rbss psect is linked into internal RAM at addresses below 3FFh, and is accessed using the direct addressing mode of the XA. This psect is cleared to all zeros by the run-time startup code before **main()** is invoked.

rbit — contains 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. The rbit psect is always linked for bit addresses in the range 100h to 1FFh Bit addresses 0 to FFh represent bits in the general register set. Bit addresses 200h to 3FFh are in the XA special function register area.

nvram — This psect is used to store **persistent** variables. It can be assigned an absolute address at link time, or by default the compiler driver will concatenate it with the bss psect. It is not cleared or otherwise modified at startup.

stack — is a psect used to specify the top of RAM memory. It does not contain any data and will always be of zero length.

heap — is another zero-length psect used to determine the highest RAM location used. The run-time heap used for allocating dynamic memory will start from this location, and grow up towards the stack (which grows downwards).

farbss — This psect will contain any variables qualified **far**. It will be assigned to the beginning of a bank.

farnvram — Any variables declared **far persistent** will be allocated in this psect, which may be located anywhere in the physical RAM address space.

5.11 Interrupt Handling in C

The compiler incorporates features allowing interrupts to be handled without writing any assembler code. The XA has multiple interrupt vectors and a priority scheme to dictate how interrupt routines are called.

5.11.1 Interrupt Functions

The function qualifier **interrupt** may be applied to a function to allow it to be called directly from a hardware or software interrupt.

The compiler will process **interrupt** functions differently to normal functions, generating code to save and restore any registers used and exit using a **reti** instead of a **ret** instruction at the end of the function. If the XAC option -STRICT is used, this 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. An example of an **interrupt** function which services the standard on-board serial port of the XA follows:

```
char            rxbuf[16];
volatile char   head, tail;
interrupt void serial_intr(void)
{
        rxbuf[head] = SBUF;
        head = (head + 1) % sizeof(rxbuf);
        if (head == tail)
                tail = (tail + 1) % sizeof(rxbuf);
        RI = 0;
}
```


5.11.2 Interrupt Priorities

The XA uses a system of priorities with interrupts. The priority level for a routine is specified with the interrupt vector and is loaded into the PSW register when the interrupt occurs. The processor is then running with that execution priority until the routine is complete. A routine of a lower priority cannot interrupt the current task if its priority is not higher than the current execution priority. The current execution priority can be changed at any point by writing a new value to the appropriate bits of the PSW register. See your XA data manual for more details.

The run-time module defines the reset vector which also has an execution priority level specified. By default the execution priority is at the highest level which implies that all routines running after reset is not interruptible. If you intend to use event or software interrupts, the execution priority must be reduced to allow the interrupt service routines to execute. This reduction in execution priority should be performed after the run-time code has completed, for example, by code in **main()**, to prevent interrupts from being serviced until the program is correctly initialised.

5.11.3 Banked Interrupts

The XA has four banks of registers that may be selected into the lower processor registers (R0-R3). These may be used in **interrupt** functions to minimise saving of registers on the stack. If the function qualifier **banked** is added to an **interrupt** function declaration, then the compiler will not generate code to save R0-R3 even if they are used. The actual bank switching is performed by loading the new PSW from the interrupt vector as described below. Note that it is entirely up to the programmer to ensure that the interrupt vector is correctly initialised to select the correct register bank, and to ensure that the **interrupt** function is not interrupted by another function using the same register bank.

If the XAC -STRICT option is used, the **banked** keyword is changed to **__banked**.

5.11.4 Interrupt Handling Macros

The standard header file <intrpt.h> contains several macros and functions which are useful when handling interrupts using C code. These are listed inTable 5 - 10 on page 141..

Table 5 - 10 Interrupt Support Macros and Functions

Macro Name	Description	Example
ei	Enable interrupts	`ei();`
di	Disable interrupts	`di();`
ROM_VECTOR	Initialise interrupt vector in ROM	`ROM_VECTOR(IV_T0, handler, IV_PSW);`
RAM_VECTOR	Initialise interrupt vector in RAM	`RAM_VECTOR(IV_T0, handler, IV_PSW);`
CHANGE_VECTOR	Alter vector in RAM	`CHANGE_VECTOR(IV_T0, handler);`
READ_RAM_VECTOR	Get current contents of vector in RAM	`iptr = READ_RAM_VECTOR(IV_T0);`

5.11.5 Enabling and Disabling Interrupts

The **di()** and **ei()** macros may be used to disable and enable maskable interrupts. It may be useful to disable interrupts while initialising or servicing I/O devices.

The macro **di()** disables interrupts by clearing the EA flag in the ICON special function register using the instruction **clr EA**. Similarly, the **ei()** macro enables interrupts by setting EA with the instruction **setb EA**.

The XA global interrupt enable flag is bit-addressable and may accessed from C code using the **bit** variable declaration:

```
static bit  EA @ 0x337;
```

which can be found in either <xa.h>, <xa-g3.h> or <xa-s3.h>. The declaration above makes it possible to test the interrupt enable state and enable or disable interrupts using C statements. For example, to enable interrupts:

```
EA = 1;
```

5.11.6 ROM_VECTOR

The **ROM_VECTOR()** macro is used to set up a "hard coded" vector in ROM which points an XA interrupt vector directly to an **interrupt** function. It takes the form:

```
ROM_VECTOR(vector, handler, psw)
```

where ***vector*** is the address of the interrupt vector and ***handler*** is the name of the **interrupt** function which will be used. ***Psw*** is the value to be copied into the PSW when the interrupt occurs.

For example, to set the serial port receive interrupt vector at address 0A0h to point to an **interrupt** function called **serial_intr()** you could write:

```
ROM_VECTOR(0xA0, serial_intr, IV_PSW)
```

The third argument to **ROM_VECTOR()** is the new PSW to be loaded when the interrupt is serviced. See the XA user manual for a full description of the PSW, but the important parts of the PSW for interrupt vectors are the system mode bit (bit 15), the execution priority (bits 8-11) and the register bank select (bits 12-13). For most applications you can use the predefined constant (from any of the standard xa header files, e.g. <xa.h>) **IV_PSW** which will set system mode, register bank 0, and an execution priority of 15, thus making the interrupt function non-interruptible. For other requirements, you can use a combination of other values, as listed in Table 5 - 11 on page 143. Note that in some XA variants (including the XA-G3 series) the interrupt priority registers implement only 3 bits for each interrupt, and allow only priority levels of 0 and 9-15. Do not attempt to use levels of 1-8 with these chips.

For example to execute the interrupt at priority 9, and using register bank 1, you could use **IV_SYSTEM+IV_PRIO09+IV_BANK1** as the third argument to **ROM_VECTOR()**. In this case the interrupt function could be declared **banked** which would prevent the saving of register R0-R3 on the stack. Note, if you use the **banked** keyword it is entirely up to you to ensure that the interrupt vector PSW setting specifies a register bank other than zero, and to ensure that only one **interrupt** function executes at any time using a given bank. You can set the PSW to any value you like, but consult the XA documentation for more information on PSW values. **ROM_VECTOR()** does not generate any code which is executed at run-time, so it can be placed anywhere in your code.

ROM_VECTOR() generates in-line assembler code, so the vector address passed to it may be in any format acceptable to the assembler. Hexadecimal interrupt vector addresses may be passed either as C style hex (0xA0) or as assembler style hex (A0h).

5.11.7 RAM-Based Interrupt Vectors

HI-TECH C supports internal RAM-based interrupt vectors which can be dynamically modified by user code, so as to point to different interrupt handlers at different points during program execution. RAM-based interrupt vectors work by setting the ROM-based interrupt vector to point to code which transfers

Table 5 - 11 Interrupt PSW Values

Value	Meaning
IV_PSW	Standard PSW setting - System mode, priority 15, bank 0
IV_SYSTEM	Set system mode
IV_PRI00	Execution priority 0 (lowest)
IV_PRI01	Execution priority 1
IV_PRI02	Execution priority 2
IV_PRI03	Execution priority 3
IV_PRI04	Execution priority 4
IV_PRI05	Execution priority 5
IV_PRI06	Execution priority 6
IV_PRI07	Execution priority 7
IV_PRI08	Execution priority 8
IV_PRI09	Execution priority 9
IV_PRI10	Execution priority 10
IV_PRI11	Execution priority 11
IV_PRI12	Execution priority 12
IV_PRI13	Execution priority 13
IV_PRI14	Execution priority 14
IV_PRI15	Execution priority 15 (highest)
IV_BANK0	Select register bank 0
IV_BANK1	Select register bank 1
IV_BANK2	Select register bank 2
IV_BANK3	Select register bank 3

control to the actual interrupt handler via an internal RAM-based pointer. The transfer of control to the user-specified interrupt handler can be achieved with minimal overhead by pushing the handler address onto the stack and then executing a **ret** instruction.

The **RAM_VECTOR()**, **CHANGE_VECTOR()** and **READ_RAM_VECTOR()** macros are used to initialise, modify and read interrupt vectors which are directed through internal RAM-based interrupt vectors in the rdata psect. These macros should only be used for vectors which need to be modifiable, so as to point at different **interrupt** functions at different points in the program. The **CHANGE_VECTOR()** and **READ_RAM_VECTOR()** macros should only be used with interrupt vectors which have been initialised using **RAM_VECTOR()**, otherwise strange things will happen.

5.11.7.1 RAM_VECTOR

The **RAM_VECTOR()** macro sets up a "soft" interrupt vector which can be modified to point to a different **interrupt** function if necessary. This is accomplished by setting up code at the vector in ROM to perform an indirect jump to the **interrupt** function, via a vector address in internal RAM. When the interrupt occurs, the code at the interrupt vector uses a **push** instruction to place the address of the handler on the stack, then executes a **ret** instruction to jump to the handler address which has just been pushed. If the interrupt vector needs to be changed, the address operand of the **push** instruction at the vector points to the "soft" vector which is in internal RAM. **RAM_VECTOR()** takes the same arguments as **ROM_VECTOR()** and can be used anywhere **ROM_VECTOR()** is used. Each use of **RAM_VECTOR()** results in an extra two bytes of initialised data in the `rdata` psect.

RAM_VECTOR() should be used only outside any functions; it generates C code and assembler code that can confuse the compiler if it appears inside a function.

5.11.7.2 CHANGE_VECTOR

The **CHANGE_VECTOR()** macro is used to modify a vector which has been set up by **RAM_VECTOR()**. This is accomplished by modifying the interrupt handler address in internal RAM. For example:

```
EA = 0;
CHANGE_VECTOR(0xA0, new_handler)
EA = 1;
```

will change the handler address used by vector 0xA0 to point to an **interrupt** function called **new_handler()**. The address of the vector word in internal RAM is found by a special symbol defined by **RAM_VECTOR()**, so **CHANGE_VECTOR()** should only be used in the same module and after **RAM_VECTOR()** has been used. The vector address should be *identical* otherwise the symbols will not match and a compile-time error will result. **CHANGE_VECTOR()** does not take a PSW value; the PSW value set with **RAM_VECTOR()** will be used.

If a vector has been modified and you want to change it back to the original value, you will need to use **CHANGE_VECTOR()** to change it back. Re-executing the code which contains the **RAM_VECTOR()** macro will not reset the vector because **RAM_VECTOR()** statically initialises the vector without generating any executable code. **CHANGE_VECTOR()** is the only vector initialisation macro which generates instructions which are actually executed at run-time, **ROM_VECTOR()** and **RAM_VECTOR()** just force initial values into the vectors.

5.11.7.3 READ_RAM_VECTOR

The **READ_RAM_VECTOR()** macro may be used to read the value of a RAM-based interrupt vector which has been set up by **RAM_VECTOR()**. It must never be used on vectors which have been initialised using **ROM_VECTOR()** as garbage will be returned. **READ_RAM_VECTOR()** can be used along with

CHANGE_VECTOR() to preserve an old interrupt handler address, set a new address and then restore the original address. For example:

```
volatile unsigned char  wait_flag;
interrupt void
wait_handler(void)
{
        ++wait_flag;
        RI = 0;
}


void
wait_for_serial_intr(void)
{
        interrupt void    (*old_handler)(void);
        EA = 0;
        old_handler = READ_RAM_VECTOR(0xA0);
        wait_flag = 0;
        CHANGE_VECTOR(0xA0, wait_handler);
        EA = 1;
        while (wait_flag == 0)
                continue;
        EA = 0;
        CHANGE_VECTOR(0xA0, old_handler);
        EA = 1;
}
```

5.11.8 Trap Interrupts

An instrinsic function **__trap()** will produce an in-line **trap** instruction, thus allowing traps to be included without the overhead of a function call. The function takes one to five arguments; the first argument is the trap number, while the remainder must be 16 bit integer values that will be placed in registers R0-R3 respectively before the **trap** instruction. The **__trap()** function returns the contents of register R0. To use this function you *must* include the header file `<xa.h>`.

For example:

```
#include<xa.h>

main()
```

```
{
        int i = 17;

        __trap(4, i, 0x1234);
}
```

will generate the following code:

```
_main:
          adds.w  r7,#-2
;x.c: 5: int i = 17;
          mov.w   [r7],#011h
;x.c: 7: __trap(4, i, 0x1234);
          mov.w   r1,#01234h
          mov.w   r0,[r7]
          trap    #04h
;x.c: 8: }
          adds.w  r7,#02h
          ret
```


The **__trap()** function is mainly intended for calling assembly language trap-handling funcions, as arguments to C functions are not in the same order as expected by the **__trap()** function. If calling a C trap-handling function using **__trap()**, use the **noregsave** and **interrupt** qualifiers so that the registers used by the function are not saved and also so that the function returns using a **reti** instruction instead of a **ret** instruction. You should preserve all general purpose registers used in the handler. Of course, you do not have to preserve any register used to pass arguments to or pass the return value from the trap handler. It is recommended to use assembler within the trap-handling function.

As an example, the following function **trap_handler()** places the one's complement of the sum of the four registers (R0 to R3) in R0. The **main()** function first sets up the trap handler using the **ROM_VECTOR()** function before calling the **__trap()** function.

```
#include <xa.h>
#include <intrpt.h>

noregsave interrupt void
trap_handler( void)
{
#asm
        add     r0,r1
        add     r0,r2
        add     r0,r3
```

```
        cpl     r0
#endasm
}

void main( void)
{
        #define TRAP_NUM 14
        unsigned char one = 0x20,   // Three args - any numbers.
                two = 0x3a,
                thr = 0x51,
                fou = 0x03;
        unsigned char result;

        ROM_VECTOR(40h+(4*TRAP_NUM),trap_handler,IV_PSW);
        result=(unsigned char)__trap(TRAP_NUM,one,two,thr,fou);
}
```

5.11.9 Predefined Interrupt Vector Names

The header file <xa.h> includes declarations for all of the standard XA interrupt vectors. These vector names may be used as the vector address argument to the **ROM_VECTOR()**, **set_vector()**, **RAM_VECTOR()**, **CHANGE_VECTOR()** and **READ_RAM_VECTOR()** macros.

The interrupt vectors defined in <xa.h> are listed in Table 5 - 13 on page 153. Interrupt vectors other than those in <xa.h> may be declared using preprocessor **#define** directives, or the vector address may be directly used with the vector macros.

5.12 Mixing C and XA Assembler Code

XA assembly language code can be mixed with C code using three different techniques.

5.12.1 External Assembly Language Functions

Entire functions may be coded in assembly language, assembled by XAS as separate .as source files 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 instruction on the XA:

```
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 XAS. The full XA assembler code for **rotate_left()** would be something like:

```
        PSECT   text,class=CODE
        GLOBAL  _rotate_left
        SIGNAT  _rotate_left,4201
        PSECT   text
_rotate_left:
        mov     R0,R3
        rl      R0,#1
        ret
```

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. In the example above, the R3 register was used for the argument to the function, and the R0 register was used for the return value. Local variable allocation, 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.

5.12.2 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
        mov     R0,#_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. 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.

5.12.3 #asm, #endasm and asm()

XA instructions may also be directly embedded in C code using the directives **#asm**, **#endasm** and the statement **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()** statement 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 (note - this example assumes direct addressing is possible - this means the that small model has been used):

```
#include        <stdio.h>
unsigned char   var;
main()
{
        var = 1;
        printf("var = 0x%2.2X\n", var);
#asm
        mov     R0,_var
        rl      R0
        mov     _var,R0
#endasm
        printf("var = 0x%2.2X\n", var);
        asm("mov  R0,_var");
        asm("rl R0,#1");
        asm("mov _var,R0");
        printf("var = 0x%2.2X\n", var);
}
```

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 `XAC -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.

5.13 Preprocessing

All C source files are preprocessed before compilation. Assembler files can also be preprocessed if the `-p` command-line option is issued, or the **Preprocess assembler files** options is ticked from within HITIDE.

5.13.1 Preprocessor Directives

XA C accepts several specialised preprocessor directives in addition to the standard directives. These are listed in Table 5 - 12 on page 151.

Macro expansion using arguments can use the **#** character to convert an argument to a string, and the **##** sequence to concatenate tokens.

5.13.2 Predefined Macros

The compiler drivers define certain symbols to the preprocessor (`CPP`), allowing conditional compilation based on chip type, memory model etc. The symbols defined are listed in Table 5 - 14 on page 154. Each symbol, if defined, is equated to 1.

5.13.3 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 - 15 on page 154. Each keyword is discussed below.

5.13.3.1 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 *backslash* "\" character when it appears as the second half of a two byte character. The **`nojis`** directive disables this special handling. JIS character handling is off by default.

5.13.3.2 The #pragma pack Directive

The XA chip requires word accesses to be aligned on word boundaries. Consequently the compiler will align all word or larger quantities onto a word boundary, including structure members. This can lead to "holes" in structures, where a member has been aligned onto the next word boundary. This behaviour can be altered with this directive. Use of the directive **`#pragma pack 1`** will prevent any padding or alignment within structures. Use this directive with caution - in general if you must access data that is not aligned on a word boundary you should do so by extracting individual bytes and re-assembling the

Table 5 - 12 Preprocessor directives

Directive	Meaning	Example
#	preprocessor null directive, do nothing	#
#assert	generate error if condition false	#assert SIZE > 10
#asm	signifies the begining of in-line assembly	#asm mov R3,[R1] #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 filename for listing	#line 3 final
#nn	(where *nn* is a number) short for #line *nn*	#20

Table 5 - 12 Preprocessor directives

Directive	Meaning	Example
#pragma	compiler specific options	See section 5.13.3 on page 150
#undef	undefines preprocessor symbol	**#undef FLAG**

data. This will result in portable code. Note that this directive must *not* appear before any system header file, as these must be consistent with the libraries supplied.

5.13.3.3 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)** to enable this checking for **printf()**. You may also use this for any user-defined function that accepts printf-style format strings. Note that the warning level must be set to -1 or below for this option to have effect.

5

5.13.3.4 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. For example, if the hardware includes an area of memory which is battery backed, it may be desirable to redirect certain variables from bss into a psect which is not cleared at startup (although this particular function is provided as a standard feature). Code and data for any of the standard C psects may be redirected using a **#pragma psect** directive. For example, if all executable code generated by a particular C source file is to be placed into a psect called altcode,the following directive should be used:

```
#pragma psect text=altcode
```

This directive tells the compiler that anything which would normally be placed in the text psect should now be placed in the altcode psect. 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 uninitialised variables which are all placed in a psect called xram, the following technique should be used:

```
---File XRAM.C
#pragma psect   bss=xram
char    buffer[20];
int     var1, var2, var3;
```

Table 5 - 13 Interrupt vectors

Name	Value	Use
IV_TRI0	0x40	Trap interrupt 0
IV_TRI1	0x44	Trap interrupt 1
IV_TRI2	0x48	Trap interrupt 2
IV_TRI3	0x4C	Trap interrupt 3
IV_TRI4	0x50	Trap interrupt 4
IV_TRI5	0x54	Trap interrupt 5
IV_TRI6	0x58	Trap interrupt 6
IV_TRI7	0x5C	Trap interrupt 7
IV_TRI8	0x60	Trap interrupt 8
IV_TRI9	0x64	Trap interrupt 9
IV_TRI10	0x68	Trap interrupt 10
IV_TRI11	0x6C	Trap interrupt 11
IV_TRI12	0x70	Trap interrupt 12
IV_TRI13	0x74	Trap interrupt 13
IV_TRI14	0x78	Trap interrupt 14
IV_TRI15	0x7C	Trap interrupt 15
IV_EX0	0x80	External interrupt 0
IV_T0	0x84	Timer 0 interrupt
IV_EX1	0x88	External interrupt 1
IV_T1	0x8C	Timer 1 interrupt
IV_T2	0x90	Timer 2 interrupt
IV_RI0	0xA0	Serial port 0 receiver interrupt
IV_TI0	0xA4	Serial port 0 transmitter interrupt
IV_RI1	0xA8	Serial port 1 receiver interrupt
IV_TI1	0xAC	Serial port 1 transmitter interrupt
IV_SWI1	0x100	Software interrupt 1
IV_SWI2	0x104	Software interrupt 2
IV_SWI3	0x108	Software interrupt 3
IV_SWI4	0x10C	Software interrupt 4
IV_SWI5	0x110	Software interrupt 5
IV_SWI6	0x114	Software interrupt 6
IV_SWI7	0x118	Software interrupt 7

Table 5 - 14 Predefined CPP Symbols

Symbol	Usage
HI_TECH_C	Always set - can be used to indicate that the compiler in use is HI-TECH C.
_XA_	Always set - can be used to indicate the code is compiled for the XA chip
SMALL_MODEL	Set for small model
MEDIUM_MODEL	Set for medium model
LARGE_MODEL	Set for large memory model.
HUGE_MODEL	Set for huge memory model

Table 5 - 15 Pragma Directives

Directive	Meaning	Example
jis	Enable JIS character handling in strings	`#pragma jis`
nojis	Disable JIS character handling (default)	`#pragma nojis`
pack	Specify structure packing values	`#pragma pack 1`
printf_check	Enable printf-style format string checking	`#pragma printf_check(printf)`
psect	Rename compiler-defined psect	`#pragma psect text=mytext`
strings	Define constant string qualifiers	`#pragma strings code`
switch	Control switch code generation	`#pragma switch direct`

Any files which need to access the variables defined in `xram.c` should **`#include`** the following header file:

```
--File XRAM.H
extern char     buffer[20];
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 not be able to use the `XAC -A` option to specify the link addresses for the new psects, instead you will need to use the `XAC -L` option to specify an extra linker option, use the linker manually or use an `HPDXA` project to compile and link your code. If you want a nearly standard configuration with the addition of only an extra psect like `xram`, you can use the `XAC -L` option to add an extra `-P` specification to the linker command. For example:

```
XAC -Bm -L-Pxram=1000h -A0,20,2000 test.obj xv.obj
```

will link test.obj and xv.obj with a standard configuration of ROM at 0h, internal RAM at 20h, external RAM at 2000h and the extra xram psect at 1000h in RAM. If you are using the HPDXA 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 ...**.

5.13.3.5 Loading Code in Large Model at Specific Addresses

When using large model, it is possible to change the name of the psect used for code (ltext) to another name, then specify a particular address to load that code in physical memory. The **#pragma psect** directive will allow you to rename the ltext psect, and you can then assign it an address at link time. For example, use

```
#pragma psect ltext=dummy
```

and all code in this module will appear in the psect dummy. You can assign an address to this psect at link time with the linker option (added into the Linker Options list in HPDXA, or supplied to XAC with the -L option) as follows:

```
-pdummy=44000h
```

This will start the psect at physical address 44000h (i.e. part-way through segment number 4).

5.13.4 The #pragma strings Directive

Any user-defined variables can be qualified by a number of type qualifiers (see Special Type Qualifiers on page 125) but constant strings (i.e. anonymous strings embedded in expressions) normally are unqualified. This means they will be put into the data segment. To control this behaviour, the **#pragma psect strings** directive allows you to specify a set of qualifiers to be applied to all subsequent constant strings. If a qualifier is specified, it will be added to any qualifiers specified previously. Using the directive without a qualifier will remove all qualifiers from any subsequent strings, i.e. restore to normal.

For example., to qualify strings with **code** you should use the example given in Table 5 - 15 on page 154. Note that all constant strings will then have type **code char *** and will not be usable where a simple **char *** type is expected.

5.13.5 The #pragma switch Directive

The compiler generates several different kinds of code for C **switch** statements. Usually the compiler will choose the smallest code for a given switch. The major methods are a *direct* switch, where a jump table is indexed by the value being switched on, and a *simple* switch, where a sequence of comparisons are done. A direct switch operates in constant time and will usually be faster on average than a simple switch, but can be quite large for a sparse set of case labels. If you have a particular need for a deterministic time switch, you can select a direct switch with this directive. A **#pragma switch**

directive will have effect only for the immediate next **switch** statement, and only if this appears in the same function. The possible arguments to this directive are **auto**, which restores behaviour to the default, **direct**, which selects a direct switch, and **simple** which selects a simple switch. See the example in Table 5 - 15 on page 154.

5.14 Linking Programs

The compiler will automatically invoke the linker unless requested to stop after producing assembler code (XAC -S option) or object code (XAC -C option).

To specify your RAM address (for variables and stack) and your ROM address (for code and initialised constants), use the XAC -A option. If the -A option is not used the compiler will ask for the appropriate ROM and RAM addresses and sizes. XAC and HPDXA by default generate *Intel* HEX files. If you use the -BIN option or specify an output file with a .bin filetype using the XAC -O option the compiler will generate a binary image instead. The file will contain code starting from the lowest initialised address in the program. For example:

```
XAC -V -Oxx.bin -A0,20,7FE0
```


will produce a binary file starting with the reset vector at address 0h, followed by the other interrupt vectors, user code, initialised data and library code.

When producing code which is to be downloaded using a debugger, the rom starting address specified should be the address of the area in RAM where the downloaded code will be located. Code which is to be run using a simulator should be compiled using the normal addresses for one of the XA variants which the simulator supports. After linking, the compiler will automatically generate a memory usage map which shows the address and size of all memory areas which are used by the compiled code. For example:

```
Memory Usage Map:
CODE: 0000H - 12E1H   12E2H (4834) bytes
IRAM: 0020H - 003FH   0020H (32) bytes
XRAM: 0100H - 0213H   0114H (276) bytes
```

More detailed memory usage information, listed in ascending order of individual psects, may be obtained by using the XAC -PSECTMAP option.

5.14.1 Replacing Library Modules

Although XA C comes with a librarian (LIBR) which allows you to unpack a library files and replace modules with your own modified versions, you can easily replace a module within a library without having to do this. If you add the source file which contains the library routine you wish to replace on the comandline list of source files, or add it to your HITIDE project, then the routine will replace the routine

in the library file with the same name. For example, if you wished to make changes to the library function **max()** which resides in the file max.c in the SOURCES directory, you could make a copy of this source file, make the appropriate changes and then compile use it as follows.

```
XAC -Bm main.c init.c max.c
```

The code for **max()** in max.c will be linked into the program rather than the **max()** function contained inthe standard libraries. Note, that if you replace an assembler module, you may need the -p option to preprocess assembler files as the library assembler files often contain C preprocessor directives.

5.14.2 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 parameters it takes 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 **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 XAC -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 XAS **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

```
XAC -S x.c
```

The resultant assembler code includes the following line:

```
SIGNAT  _widget,8249
```

The **SIGNAT** pseudo-op tells the assembler to include a record in the .obj file which associates the value 8249 with symbol **_widget**. The value 8249 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);
```

A different signature will be generated and the linker will report a signature mismatch which will alert you to the possible existence of incompatible calling conventions.

5.14.3 Linker-Defined Symbols

In order for the run-time code to clear the bss and rbss psects and copy the data and rdata psects, it must determine the load address, link address and size of these psects. This is achieved using special linker-generated symbols.

The link address of a psect can be obtained from the value of a global symbol with name **__L*name*** 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 **__H*name***. 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 **__B*name***.

5.15 Optimising Code for the XA

Care needs to be taken to avoid writing code which will be large or inefficient. To improve execution speed and reduce code size, some or all of these suggestions can be used:

- In the medium and large models, variables which are critical to performance should be declared **near**, which places them in directly addressable RAM.
- Chose a memory model which is applicable to the application. If you have only modest RAM requirements, then use the small model.
- Use 8-bit quantities (**char** or **unsigned char**) where appropriate rather than 16-bit quantities, as this consumes less space and can be accessed more quickly if using an 8-bit external bus.

5.16 Standard I/O Functions and Serial I/O

A number of the standard I/O functions are provided in the C library with the XA 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 - 16 on page 159. More details of these functions are in the “Library Functions” chapter. By default any characters written or read using these functions will be sent or received via the on-board serial port. If however the code is run under the Lucifer debugger, then the host portion of the debugger will act as a dumb terminal. This behaviour is determined by two library

Table 5 - 16 Supported STDIO Functions

Function name	Purpose
`puts(char * s)`	Writes a string to stdout, appends newline
`char * gets(char * buf)`	Gets a line of text from stdin to buf, removes newline
`printf(char * s, ...)`	Formatted printing to stdout
`putchar(int c)`	Puts a single character to stdout
`scanf(char *, ...)`	Reads formatted input from stdin
`sprintf(char * buf, char * s, ...)`	Writes formatted text to buf
`sscanf(char * buf, char * s, ...)`	Reads formatted text from buf
`vprintf(char * s, va_arg list)`	Version of printf taking argument list
`vscanf(char * s, va_arg list)`	Version of scanf taking argument list
`vsprintf(char * buf, char * s, va_arg list)`	Version of sprintf taking argument list
`vsscanf(char * buf, char * s, ...)`	Version of sscanf taking argument list

modules, the source for which is found in the SOURCES directory. The modules are `getch.c` which contains the functions **`getch()`**, **`putch()`**, **`getche()`**, and **`kbhit()`**, and `serial.c` which has lower-level functions to drive the serial port. If you wish to send or receive characters via some other means (e.g. to an LCD display) you should modify one or other of these modules. If `getch.c` is unmodified, then the debugger will continue to manage standard I/O when used.

To replace one of these modules with your own version, copy the source code from the SOURCES directory to a file in your working directory, make whatever changes are needed, and include this file in your project - either in the **Source files list** in HPDXA or on the command line to XAC. Your module will then be used rather than the library module. You must however retain all the functions present in the module, even if some are unused (in which case they can be empty functions). Failure to do so may lead to multiply defined symbol messages at link time.

The XA Macro Assembler

The HI-TECH Software XA Macro Assembler assembles source files for the Philips XA family of microprocessors. This chapter describes the usage of the assembler and the directives (assembler pseudo-ops and controls) accepted by the assembler.

For a description of the available special function registers and instructions refer to the appropriate processor handbook.

The HI-TECH assembler package includes a linker, librarian, cross reference generator and an object code converter.

6.1 Assembler Usage

The assembler is called ASXA and is available to run under the UNIX and MS-DOS operating systems. 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 is:

```
asxa [ 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 162, and a full description of each option follows.

6.2 Assembler Options

The command line options recognised by ASXA are as follows:

-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 listed on the command line, will contain raw cross reference information. The cross reference utility CREF must then be run to produce the formatted cross reference listing.

Table 6 - 1 ASXA Assembler options

Option	Meaning	Default
-C	Produce cross-reference	No cross reference
-D	Produce debug summary	
-D*symb*	Define a symbol	
-F*length*	Specify listing form length	66
-I	List macro expansions	Don't list macros
-L*listfile*	Produce listing	No listing
-O*outfile*	Specify object name	srcfile.OBJ
-Q	Quick assembly	Optimized assembly
-S	No size error messages	
-U	No undef'd symbol messages	
-V	Produce line number info	No line numbers
-W*width*	Specify listing page width	80 or 132
-X	No local symbols in OBJ file	

`-D` The `-D` option directs the assembler to generate a summary of how many **`nop`** instructions and pad bytes where used in the assembly of the file.

`-D`*`symb`* The `-D`*`symb`* option is used to define a symbol. This option may take one of two forms:

`-D`*`symb`* which is equivalent to:

```
symb  EQU  1
```

or `-D`*`symb`*`=`*`numeric`* which is equivalent to:

```
symb  EQU  numeric
```

`-F`*`length`* 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.

`-I` If the **`GEN`** assembler control is being used to generate macro expansions, this option overrides any **`NOLIST`** assembler controls and forces listing of macro expansions and unassembled conditionals.

`-L`*`listfile`* 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`*`outfile`* 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 and explicit filename for the object file.

`-Q` The default mode of operation of the assembler is to iterate over the source code until the smallest possible code is produced, by optimizing jumps. If the `-Q` option is used then only two passes over the source code will be made, thereby speeding up assembly. This may result in **`nop`** instructions being generated in the code where an optimization was performed on the second pass but not the first.

`-S` If a byte-size memory location is intialized 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 these messages.

`-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.

`-W`*`width`* 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 if the listfile is a device (terminal, printer etc.) or 132 if it is a file.

`-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 XA Assembly Language

The source language accepted by the HI-TECH Software XA Macro Assembler is described below. All opcode mnemonics and operand syntax are strictly as described in the Philips XA User's Guide.

6.3.1 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.

6.3.2 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 on page 164.

Table 6 - 2 ASXA Numbers and bases

Radix	Format
Binary	digits 0 and 1 followed by **B**
Octal	digits 0 to 7 followed by **O**, **Q**, **o** or **q**
Decimal	digits 0 to 9 followed by **D**, **d** or nothing
Hexadecimal	digits 0 to 9, A to F preceded by **0x** *or followed by* **H** *or* **h**

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.

Real numbers are accepted in the usual format for **DF** and **DD** directives only. The exponent and mantissa of a real number must be decimal. When using the **DF** directive, real numbers are stored in IEEE 32 bit format; when using the **DD** directive, they are stored in the fast double format. A real number should include a decimal point, but the exponent and sign are optional. If the exponent is present, it should follow the mantissa without any intervening white space.

6.3.3 Delimiters

All numbers and identifiers must be delimited by white space, non-alphanumeric characters or the end of a line.

6.3.4 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.

6.3.5 Identifiers

Identifiers are user-defined symbols representing memory locations or numbers. A symbol may contain any number of characters drawn from the alphabetics, numerics and the special characters *dollar* "$", *question mark* "?" and *underscore*"_". The first character of an identifier may not be numeric. The case of alpahabetics is significant, e.g. **Fred** is not the same symbol as **fred**.

6.3.5.1 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 bits, 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.

6.3.5.2 SFR Names

Since the set of SFRs in any given XA variant may differ from other devices, no predefined SFR names are provided in the assembler. SFR names should be defined using the **EQU** directive.

6.3.5.3 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.

6.3.5.4 Location Counter

The current location within the active program section is accessible via the dollar symbol **$**.

6.3.5.5 Register Symbols

All registers are available by using their standard names, e.g. R0 r3h etc. The case of register names is not significant. It is also possible to equate a symbol to a register, and from then on that symbol will be interpreted as equivalent to the register name. Equated symbols used in this way *are* case sensitive. A register name may not be used for any other purpose, e.g. as a macro argument name. The special symbol **SP** is defined and is equivalent to **R7**.

6.3.6 Strings

A string is a sequence of characters not including *carriage return* or *newline*, enclosed within matching quotes. Either single quotes "**'**" or double quotes "**"**" 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.

6.3.7 Temporary Labels

The assembler implements a system of temporary labels (as distinct from the local labels used in macros) which relieves the programmer from creating new labels within a block of code. A temporary label is defined as a numeric string, 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 **f**orward from the current location, and conversely a "**b**" will cause the assembler to look **b**ackward for the label.

For example:

```
entry:
        mov     r0,ploc
1:
        mov     a,@r0
        jz      1f      ;end of string
        inc     r0
        cjne    a,r2,1b
        sjmp    2f      ;found it
1:
        clr     a       ;return zero
        ret
2:
        dec     r0
        mov     a,r0    ;return pointer

        ret
```

Note that even though there are two **1:** labels, no ambiguity occurs, since each is referred to uniquely. The **cjne 1b** refers to a label further back in the source code, while **jz 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 Expressions

Expressions are made up of numbers, symbols, strings and operators. The available operators are listed in Table 6 - 3 on page 167, in order of precedence. 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.

6.3.9 The Bit Operator

The bit operator (the *dot* symbol "**.**") is provided as an easy means of calculating bit addresses from byte addresses. It does not, however, create a symbol that is restricted to use with bit instructions. The bit operator takes a byte address on the left, and a bit number on the right. These values are checked at assembly time if possible, or at link time, for correctness. The resulting value is the equivalent bit address. This value may, however, be used in any context. If an error is detected at link time with a bit address then a message of the form:

Table 6 - 3 Operators

Operator	Purpose	Precedence
NUL	Test for null argument	8
^	Exponentation	7
*** ,/ , MOD**	multiply divide modulus	6
SHR, SHL	shift right, shift left	6
ROR, ROL	rotate right, rotate left	6
+, -, .	plus, minus (unary or binary), bit	5
HIGH	high byte of word expression	5
LOW	low byte of word expression	5
SEG	segment part of address	5
EQ, NE, GT, GE, LT, LE	Relational operators	4
=, <>, >, >=, <, <=	Relational operators	4
NOT	bitwise inversion	3
AND	bitwise conjunction	2
OR	bitwise disjunction	1
XOR	exclusive OR	1

```
bit address range check failed
```

or

```
bit number range check failed
```

will be issued. An example of the use of the bit operator is:

```
P0     EQU    430h
AD0    EQU    P0.0
```

6.3.10 Statement Format

Legal statement formats are shown in table Table 6 - 4 on page 167. 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

Table 6 - 4 ASXA Statement formats

label:	***opcode***	***operands***	***;comment***
name	***pseudo-op***	***operands***	***;comment***
;comment only			

label, if present, is 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 Addressing Modes

The assembler recognizes all standard XA addressing modes. Refer to the *Philips* document *16-bit 80C51XA Microcontrollers (eXtended Architecture) Data Handbook*, document order number 9397-750-01784 for full information on opcodes and addressing modes.

6.3.12 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. 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 169 for full information. The assembler associates no significance to the name of a psect.

6.3.13 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 Table 6 - 5 on page 169, and detailed below.

6.3.13.1 PUBLIC

The **PUBLIC** directive takes a comma separated list of symbols defined in the current module and which are to be accessible to other modules at link time. Example:

```
PUBLIC      lab1,lab2,lab3
```

6.3.13.2 EXTRN

This is the complement of **PUBLIC**; it declares symbols which may then be referenced even though they are defined in another module. Example:

```
EXTRN       lab1,lab2,lab3
```

6.3.13.3 GLOBAL

GLOBAL is a combination of **PUBLIC** and **EXTRN**; it declares a list of symbols which, if defined within the current module, are made public, otherwise are made external. Example:

```
GLOBAL      lab1,lab2,lab3
```

Table 6 - 5 ASXA Directives (pseudo-ops)

Directive	Purpose
PUBLIC	Make symbols accessible to other modules
EXTRN	Allow reference to other modules symbols
GLOBAL	Public or extrn as appropriate
END	End assembly
PSECT	Declare or resume program selection
ORG	Set location counter
EQU	Define symbol value
SET	Re-define symbol value
DB	Define constant byte(s)
DW	Define constant word(s)
DF	Define constant real(s)
DS	Reserve storage
IF	Conditional assembly
ELSIF	Alternate conditional assembly
ELSE	Alternate conditional assembly
ENDIF	End conditional assembly
MACRO	Macro definition
ENDM	End macro definition
LOCAL	Define local tabs
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
EXITM	Terminate macro expansion
SIGNAT	Define function signature

6.3.13.4 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
```

6.3.13.5 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 on page 170 and detailed

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
GLOBAL	Psect is global (default)
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
ALIGN	Align all labels in psect to specified boundary
SIZE	Max size of psect
LIMIT	Limit of psect
BIT	Psect allocated in one bit units, rather than bytes
CLASS	Specify class name for psect
SPACE	Specify space identifier for psect

- The **bit** flag defines the current psect as being bit-addressable. Any storage allocated in a psect which uses the **bit** psect flag will be in bits, not bytes. For example, **DS 4** in a bit psect will reserve 4 bits of storage.
- 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.
- **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.
- 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.
- 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.
- 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. Where there are two **local** psect directives in the one module, they refernce the same psect. A local psect may not have the same name as any global psect,

even one in another module.

- 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 **limit** flag allows an upper address limit to be specified for the psect, e.g. **limit=8000h**. This will be checked by the linker after psects have been combined from all modules.
- The **reloc** flag allows specification of a requirement for alignment of the psect on a particular boundary, e.g. **reloc=2** would specify that this psect must start on an even address.
- The **align** directive specifies that all labels defined in this psect should be aligned onto an address that is a multiple of the value supplied, e.g. **align=2** would force all labels to an even address. This is required for any psects which will have XA executable code.
- 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 **space** flag allows an address space number to be associated with the psect. At link time, psects in the same address space will be checked to ensure that there are no overlaps. The checking is based on the *load* address of the psect (see the linker manual for a description of link and load addresses). Psects that are in different address spaces will not be reported if they overlap.

Some examples of the use of the PSECT directive follow:

```
PSECT  fred
PSECT  bill,size=100h,global
PSECT  joh,abs,ovrld,class=CODE
```


6.3.13.6 ORG

ORG changes the value of the location counter within the current psect. 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
```

6.3.13.7 EQU and SET

This pseudo-op defines a symbol and equates its value to an expression. For example:

```
assembly EQU 123h
```

The identifier **assembly** will be given the value 123h. **EQU** is legal only when the symbol has not previously been defined. The operand may also be a register name, in which case the symbol will become a synomym for the register.

SET is identical to **EQU** except that it may be used to re-define a symbol.

6.3.13.8 DB and DW

These directives initialize storage, as bytes or words respectively. The argument to each is a list of expressions, each of which will be assembled into one byte or word. **DB** may also take a multi-character string as an argument. 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,"A string",0
        DW   23*10,alabel,0,'a'
```

6.3.13.9 DF and DD

DF initializes 4 bytes of memory as real numbers. Each number will be stored in IEEE 32 bit format, low byte first.

```
pi:   DF    3.14159
      DF    3.3,3e10,-23
```

DD initializes 8 bytes of memory as real numbers. Each number will be stored in fast double format, low byte first, as described in 5.4.6 on page 122.

```
pi:   DD    3.14159
      DD    3.3,3e10,-23
```

6.3.13.10 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
xlabel: DS   2+3
```

6.3.13.11 IF, ELSE, ELSEIF and ENDIF

These directives implement conditional assembly. The argument to **IF** should be an absolute expression. If it is non-zero, then the code following it up to the next matching **ELSE** or **ENDIF** will be assembled. If the expression is zero then the code up to the next matching **ELSE** or **ENDIF** 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      some_symbol
mov     R1,[R0]
ELSE
movc    R1,[R0+]
ENDIF
```

In this example, if ***some_symbol*** is non-zero, the first **mov** instruction will be assembled but not the second. Conversely if ***some_symbol*** is zero, the **movc** will be assembled but not the first **mov** will not. Conditional assembly blocks may be nested.

Instead of **ELSE**, an **ELSEIF** may be used to test a further condition. There may be a sequence of **ELSEIF** statements after an **IF**, but there must be only one **ELSE** before the corresponding **ENDIF**. For example:

```
IF          a_value
mov         r1[,[r0]
ELSEIF      b_value
mov         r2,[r0]
ELSE
mov         r3,[r1]
ENDIF
```

6.3.13.12 MACRO and ENDM

These directives provide for the definition of macros. The **MACRO** directive should be preceded by the macro name and 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.

The following is an example which defines the **swapr** macro:

```
swapr  MACRO  reg1, reg2          ;exchange registers
       mov    r0,r@reg1@          ;save reg1
       mov    r@reg1,r@reg2       ;reg2 --> reg1
       mov    r@reg2,r0           ;restore reg2
       ENDM
```

The macro invocation **swapr 3,4** would expand to:

```
       mov    r0,r3
       mov    r3,r4
       mov    r4,r0
```

The **@** character may be used to delimit an argument used in the coding of the macro, thus permitting the concatenation of macro parameters with other text, but is removed in the actual macro expansion. The

@ character need not be used if *commas* or *spaces* are delimiting the argument, but should be used at both ends if no other delimiters are available.

The **nul** operator may be used within a macro to test a macro argument. For example:

```
IF nul reg2
...
ELSE
...
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* "**;;**".

6.3.13.13 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:

```
copy   MACRO        src,dst,cnt
       LOCAL        loop
       mov          r0,src
       mov          r1,dst
       mov          r2,#cnt
loop:  mov          [R1+],[R0+]
       djnz         r2,loop
       ENDM
```

defines a macro **copy** which when invoked as **copy #inbuf,#procbuf,32** expands to:

```
       mov          r0,#inbuf
       mov          r1,#procbuf
       mov          r2,#32
??0001:mov          [R1+],[R0+]
       djnz         r2,??0001
```

If invoked a second time, the label **loop** would expand to **??0002**.

6.3.13.14 REPT

The REPT directive temporarily defines an unnamed macro then expands it a number of times as determined by its argument. For example:

```
mov         r0,#zbuf
clr         r1l
REPT        3
mov         [R0],r1l
inc         r0
ENDM
```

expands to:

```
mov         r0,#zbuf
clr         r1l
mov         [r0],a
inc         r0
mov         [R0],a
inc         r0
mov         [R0],a
inc         r0
```

6.3.13.15 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 or **IRPC** it is each character in one argument. For each repetition the argument is substituted for one formal parameter.

For example:

```
IRP    arg,lab1,lab2,#23
mov    [r0],arg
inc    r0
ENDM
```

expands to:

```
mov    [r0],lab1
inc    r0
mov    [r0],lab2
inc    r0
mov    [r0],#23
inc    r0
```

The **IRPC** directive is similar, except it substitutes one character at a time from a string of non-space characters. For example:

```
        IRPC   arg,ABC
        LOCAL  lab
        cjne   a,#'arg',lab
        ljmp   case_&arg
lab:
        ENDM
```

expands to:

```
        cjne   a,#'A',??0000
        ljmp   case_A
??0000:
        cjne   a,#'B',??0001
        ljmp   case_B
??0001:
        cjne   a,#'C',??0002
        ljmp   case_C
??0002:
```

6.3.13.16 .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.

6.3.14 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.

6.3.15 Assembler Controls

Control lines may be included in the assembler source to control such things as listing format. Each control line starts with a *dollar* "**$**" character which is followed by a white space-separated list of control keywords. These keywords have no significance anywhere else in the program. Some keywords may have a parameter after them, which is always enclosed in parentheses. Most control keywords have a positive and a negative form. All have two letter abbreviations, the negative form is constructed by prefixing the keyword or the abbreviation with **NO**.

A list of keywords is given in Table 6 - 7, and each is described further below..

Table 6 - 7 : ASXA Assembler controls

Control name	Abbreviation	Default
COND/NOCOND	C0/NOCO	CO
EJECT	EJ	
GEN/NOGEN	GE/NOGE	NOGE
INCLUDE(pathname)	IC	
LIST/NOLIST	LI/NOLI	LI
PAGELENGTH(n)	PL	PL(66)
PAGEWIDTH(n)	PW	PW(120)
SAVE/RESORE	SA/RS	
TITLE(string)	TT	
XREF/NOXREF	XR/NOXR	NOXR

6.3.15.1 COND

When **COND** is in effect, lines of code not assembled because of conditional assembly will be listed. See also the **NOCOND** control.

6.3.15.2 EJECT

EJECT causes a new page to be started in the listing. A *control-L* (form feed) character will also cause a new page when encountered in the source.

6.3.15.3 GEN

When **GEN** is in effect the code generated by macro expansions will be listed. See also the **NOGEN** control.

6.3.15.4 INCLUDE(pathname)

This control causes the file specified by `pathname` to be textually included at that point in the listing. The **INCLUDE** control must be the last control keyword on the line.

6.3.15.5 LIST

If the listing was previously turned off using the **NOLIST** control, the **LIST** control on its own will turn the listing on. See also the **NOLIST** control.

6.3.15.6 NOCOND

Any conditional code will not be included in the listing output. See also the **COND** control.

6.3.15.7 NOGEN

NOGEN disables macro expansion in the listing file. The macro call will be listed instead. See also the **GEN** control.

6.3.15.8 NOLIST

This control turns the listing output off from this point onwards. See also the **LIST** control.

6.3.15.9 NOXREF

NOXREF will disable generation of the raw cross reference file. See also the **XREF** control.

6.3.15.10 PAGELENGTH(*n*)

This control keyword specifies the length of the listing form. The default is 66 (11 inches at 6 lines per inch).

6.3.15.11 PAGEWIDTH(*n*)

PAGEWIDTH allows the listing line width to be set.

6.3.15.12 RESTORE

RESTORE pops the top of the stack off into the flags. It may be used to selectively control listing inside macros. See also the **SAVE** control.

6.3.15.13 SAVE

SAVE pushes the current state of the **LIST**, **COND** and **GEN** flags onto a stack. See also the **RESTORE** control.

6.3.15.14 TITLE(*string*)

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.

6.3.15.15 XREF

XREF is equivalent to the command line option `-C`, 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. See also the **NOXREF** control.

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 Psects

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 psect, 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 *psects*. These psects represent logical groupings of certain types of code bytes in the program. In general the compiler will produce code in three basic types of psects, although there will be several different types of each. The three basic kinds are `text` psects, containing executable code, `data` psects, containing initialised data, and `bss` psects, 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 the **`PSECT`** directive **`class=`** in assembler code. See *The Macro Assembler* chapter for more information on **`psect`** options.

7.5 Global Symbols

The linker handles only symbols which have been declared as **`GLOBAL`** to the assembler. The code generator generates these assembler directives whenever it encounters global C objects. At 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:

```
hlink[1] 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 183 and discussed in the following paragraphs.

Table 7 - 1 Linker Options

Option	Effect
-A*class=low-high,...*	Specify address ranges for a class
-C*x*	Call graph options
-C*psect=class*	Specify a class name for a global psect
-C*baseaddr*	Produce binary output file based at *baseaddr*
-D*class=delta*	Specify a class delta value
-D*symfile*	Produce old-style symbol file
-E*errfile*	Write error messages to *errfile*
-F	Produce .obj file with only symbol records
-G*spec*	Specify calculation for segment selectors
-H*symfile*	Generate symbol file
-H+*symfile*	Generate enhanced symbol file
-I	Ignore undefined symbols
-J*num*	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
-M*mapfile*	Generate a link map in the named file
-O*outfile*	Specify name of output file

1. In earlier versions of HI-TECH C the linker was called LINK.EXE

Table 7 - 1 Linker Options

Option	Effect
-P*spec*	Specify psect addresses and ordering
-Q*processor*	Specify the processor type (for cosmetic reasons only)
-S	Inhibit listing of symbols in symbol file
-S*class=limit[,bound]*	Specify address limit, and start boundary for a class of psects
-U*symbol*	Pre-enter symbol in table as undefined
-V*avmap*	Use file *avmap* to generate an *Avocet* format symbol file
-W*warnlev*	Set warning level (-10 to 10)
-W*width*	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 -A*class=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 -C*x*

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 -C*psect=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 -D*class=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 -D*symfile*

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 -E*errfile*

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 -G*spec*

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=** directive 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 -H*symfile*

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 -J*errcount*

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 -M*mapfile*

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 191.

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* value, or class.

7.7.19 -O*outfile*

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 -P*spec*

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 -Q*processor*

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 -S*class*=*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 -U*symbol*

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 -V*avmap*

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 -W*num*

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 of *num* >= 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\l.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 objects of type **char** 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 s**printf()**. 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 takes 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 affect 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 are 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
-P*width*	specify page width
-W	suppress non-fatal errors

The key letters are listed inTable 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 object 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 198. 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:

Table 7 - 4 Objtohex Options

Option	Meaning
-A	Produce an ATDOS .atx output file
-B*base*	Produce a binary file with offset of *base*. Default file name is l.obj
-C*ckfile*	Read a list of checksum specifications from *ckfile* or standard input
-D	Produce a COD file
-E	Produce an MS-DOS .exe file
-F*fill*	Fill unused memory with words of value *fill* - 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
-P*stk*	Produce an output file for an *Atari* ST, with optional stack size
-R	Include relocation information in the output file
-S*file*	Write a symbol file into *file*
-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

```
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.

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 199. Each option is described in more

Table 7 - 5 Cref Options

Option	Meaning
-F*prefix*	Exclude symbols from files with a pathname or filename starting with *prefix*
-H*heading*	Specify a heading for the listing file
-L*len*	Specify the page length for the listing file
-O*outfile*	Specify the name of the listing file
-P*width*	Set the listing width
-S*stoplist*	Read file *stoplist* and ignore any symbols listed.
-X*prefix*	Exclude any symbols starting with the given *prefix*

detail in the following paragraphs.

7.12.1 -F*prefix*

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 -H*heading*

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 -L*len*

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 -O*outfile*

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 -P*width*

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 -S*stoplist*

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 -X*prefix*

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 `-Xxyz` 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 Cromwell

The `CROMWELL` utility converts code and symbol files into different formats. The formats available are shown in Table 7 - 6.

Table 7 - 6 Format Types

Key	Format
`cod`	*Bytecraft* COD file
`coff`	COFF file format
`elf`	ELF/DWARF file
`eomf51`	Extended OMF-51 format
`hitech`	HI-TECH Software format
`icoff`	ICOFF file format
`ihex`	*Intel* HEX file format
`omf51`	OMF-51 file format
`pe`	P&E file format
`s19`	*Motorola* HEX file format

The general form of the `CROMWELL` command is:

```
cromwell options input_files -okey output_file
```

where *options* can be any of the options shown in Table 7 - 7. *Output_file* (optional) is the name of the output file. The *input_files* are typically the HEX and SYM file. CROMWELL automatically searches for the SDB files and reads those if they are found. The options are further described in the following paragraphs.

7.13.1 -Pname

The -P options takes a string which is the name of the processor used. CROMWELL may use this in the generation of the output format selected.

7.13.2 -D

The -D option is used to display to the screen details about the named input file in a readable format.

Table 7 - 7 Cromwell Options

Option	Description
-P*name*	Processor name
-D	Dump input file
-C	Identify input files only
-F	Fake local symbols as globals
-O*key*	Set the output format
-I*key*	Set the input format
-L	List the available formats
-E	Strip file extensions
-B	Specify big-endian byte ordering
-M	Strip underscore character
-V	Verbose mode

The input file can be one of the file types as shown in Table 7 - 6.

7.13.3 -C

This option will attempt to identify if the specified input files are one of the formats as shown in Table 7 - 6. If the file is recognised, a confirmation of its type will be displayed.

7.13.4 -F

When generating a COD file, this option can be used to force all local symbols to be represented as global symbols. The may be useful where an emulator cannot read local symbol information from the COD file.

7.13.5 -Okey

This option specifies the format of the output file. The *key* can be any of the types listed in Table 7 - 6.

7.13.6 -Ikey

This option can be used to specify the default input file format. The *key* can be any of the types listed in Table 7 - 6.

7.13.7 -L

Use this option to show what file format types are supported. A list similar to that given in Table 7 - 6 will be shown.

7.13.8 -E

Use this option to tell CROMWELL to ignore any filename extensions that were given. The default extension will be used instead.

7.13.9 -B

In formats that support different endian types, use this option to specify big-endian byte ordering.

7.13.10 -M

When generating COD files this option will remove the preceeding *underscore* character from symbols.

7.13.11 -V

Turns on verbose mode which will display information about operations CROMWELL is performing.

7.14 Memmap

MEMMAP has been individualized for each processor. The *MEMMAP* program that appears in your BIN directory will conform with the following criteria; *XX*map.exe where *XX* stands for the processor type. From here on, we will be referring to this application 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.

7.14.1 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 - 8 on page 203. *File* is the name

Table 7 - 8 Memmap options

Option	Effect
-P	Print psect usage map
-W*wid*	Specifies width to which address are printed

of a map file. Only one map file can be processed by *MEMMAP*.

7.14.1.1 -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.

7.14.1.2 -W*wid*

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.

Lucifer source level debugger

Lucifer is a source level remote debugger for use with the HI-TECH C compilers. It consists of a program which runs on a host machine (usually MS-DOS or UNIX) and communicates with an XA-based microcontroller system via a serial line. The host program provides the user interface, including source code display, disassembly, displaying memory etc. The target system must have logic to read and write memory and registers, and implement single stepping. With each version of Lucifer a small program is provided which can be compiled and placed in a ROM in a target system to implement these features. The standard host program is set up to communicate with this ROM program via a serial line.

8.1 Using Lucifer

To use Lucifer you will need to have the compiler generate a symbol file, with symbol name, line number and file name symbols included. This can be produced using the XAC **-G** option. If you use the **-H** option you will get a symbol file which can be used by Lucifer, but which does not contain any source code level information.

You can also use HPDXA to produce HEX and symbol files suitable for use with Lucifer. The LUCIFER directory contains *luctest.prj*, a sample project file which you can use as a guide to compiling your own programs with HPDXA. See the Using HPDXA chapter for more details. The basic option needed is **Options/Source level debug info**. Lucifer can be invoked from the **Run/Download ...** menu item. Otherwise once you have produced HEX and symbol files, invoke Lucifer as follows:

```
lucxa -sSPEED -pCOMn test
```

If you are using an MS-DOS system, *COMn* should be COM1 or COM2. Lucifer will access a standard serial port addressed as either port. For UNIX simply specify the name of the serial port connected to your target, for example *-p/dev/tty006*.

The default baud rate is 38400 for both MS-DOS and UNIX. The default serial port is *COM1* for MS-DOS and */dev/ttya* for UNIX. The -s (speed) and -p (port) options may be used to access ports other than the default. For UNIX */dev* may be left off the device name and will automatically added, thus *-ptty0* and *-p/dev/tty0* will access the same device. For example, under MS-DOS, *lucxa -s9600 -pcom2* will access COM2 at 9600 baud.

New default speed and port values may be set using the environment variable **LUCXA_ARGS**. **LUCXA_ARGS** may specify any mixture of valid Lucifer '-' options except filename options. For example, if you want the default options to be 4800 baud on port COM2, add the following line to your AUTOEXEC.BAT file:

```
SET LUCXA_ARGS=-s4800 -pCOM2
```

In addition to the speed and port options Lucifer takes two optional arguments which are, in order of appearance, the name of the symbol file to use and the name of the *.hex* or *.bin* file to download. If no download file is specified, Lucifer will automatically search for *.sym*, *.hex* and *.bin* files which the same base name as the symbol name given. Thus the command:

```
lucxa test
```

would automatically locate and use *test.sym* and *test.hex* or *test.bin*. If you do not want to autoload your HEX or BIN file, give your symbol file a different base name to your HEX file.

When downloading binary files, Lucifer normally prompts for the download address. When downloading directly from the command line you can override this prompting by adding the option *-Baddr[:end]* to the LUCXA command line. For example, if you want to download a file *test.bin* at address $2000, you could use the command:

```
lucxa -b2000 test
```

The optional *:end* value is the address at which the download should be terminated. For example, if you want to load the first $2000 bytes of ***test.bin*** from address $4000 to address $6000, use the command:

```
lucxa -b4000:6000 test
```

Lucifer should announce itself, then attempt to communicate with the target. If successful it prints a message sent by the target, identifying itself, e.g:

```
XA Lucifer Monitor (+ diagnostics) V4.3
```

Lucifer will then display a prompt `:` and wait for commands. For a list of commands, type `?` and press return. Note that all commands should be in lower case.

8.2 Symbol Names In Expressions

Where Lucifer commands take numeric values or addresses as arguments; symbol names, register names and line numbers may be used. Symbols should be entered in exactly the same case as they were defined in the source code. Note that Lucifer cannot access auto variables or parameters by name. Where an expression is required, it may be of the forms in Table 8 - 1 on page 207.

By default, in the **b** (breakpoint) command any decimal number will be interpreted, as a line number. However, in the **u** (unassemble) command any number will be interpreted, by default, as a hex number representing an address. These assumptions can always be overridden by using the colon or dollar prefixes. When entering a symbol, it is not necessary to type the underscore prepended by the C compiler. However, when printing out symbols the debugger will always print the underscore. Any register name may also be used where a symbol is expected.

Table 8 - 1 Lucifer expression forms

Expression form	Example
symbol_name	main
symbol+hexnum	barray+20
$hexnum	$2000
:linenumber	:10
regname	A5

8.2.1 Auto Variables and Parameters

Auto variables and parameters cannot be accessed by name with Lucifer. To examine the contents of an auto variable or parameter, the best approach is to disassemble (with the **u** command) a line of code referencing the variable, and dump the corresponding memory location (e.g. **sp+4**).

8.3 Lucifer Command Set

Lucifer recognizes the commands listed in Table 8 - 2 on page 208.

8.3.1 The B command: set or display breakpoints

The b command is used to set and display breakpoints. If no expression is supplied after the b command, a list of all currently set breakpoints will be displayed. If an expression is supplied, a breakpoint will be set at the line or address specified. If you attempt to set a breakpoint which already exists, or enter an expression which Lucifer cannot understand, an appropriate error message will be displayed. Note: by default, any decimal number specified will be interpreted as a line number. If you want to specify an absolute address, prefix it with a dollar sign. For example:

```
: b 10
Set breakpoint at _main+$28
:
```

Breakpoints can also include a semicolon separated list of Lucifer commands, which will be executed when the breakpoint is encountered. This makes it possible to create breakpoints which stop, display a value and then restart execution. For example, the command:

```
: b 10 @f pi;g
```

creates a breakpoint which stops, displays the value of global variable pi and then continues execution.

Table 8 - 2 Lucifer Command Set

Command	Meaning
B *[line|addr]*	Set or display breakpoints
C	Display instruction at PC
D *[addr [addr]]*	Display memory contents
E *fn|file*	Examine C source code
G *[addr]*	Commence execution
I	Toggle instruction trace mode
L *file*	Load a HEX file
M *addr val1 [val2 ...]*	Modify memory
Q	Exit to operating system
R *[breakpnt]*	Remove breakpoints
S	Step one line
T	Trace one instruction
U *[addr]*	Disassemble machine instructions
W *file addr length*	Upload binary
X *[reg1 val1 [reg2 val2 ...]]*	Examine or change registers
@ type [indirection]expr	Display C variables
. [breakpoint]	Set a breakpoint and go
; [line]	Display from a source line
=	Display next page of source
-	Display previous page of source
/ [string]	Search source file for a string
! command	Execute a DOS command

8.3.2 The C command: display instruction at PC

The **c** command is used to display the assembler instruction and C source line addressed by the current value of the program counter. This is useful if you have been using other Lucifer commands and aren't quite sure where in the program the program counter is pointing. For example:

```
: c
10:   printf("answer = %d\n",j);
_main+$22   MOV   r0,_j
```

8.3.3 The D command: display memory contents

The **d** command is used to display a hex dump of the contents of memory on the target system. If no expressions are specified, 16 bytes are dumped from the address reached by the last **d** command. If one address is specified, 16 bytes are dumped from the address given. If two addresses are specified, the contents of memory from the first address to the second address are displayed. Dump addresses given can be symbols, line numbers, register names or absolute memory addresses.

8.3.4 The E command: examine C source code

The **e** command is used to examine the C source code of a function or file. If a function name is given, Lucifer will locate the source file containing the function requested and display from just above the start of the function. If a file name is given, Lucifer will display from line 1 of the requested file. For example:

```
: e main
2:
3:int value, result;
4:
5:main()
6:{
7:    scanf("%d",&value);
8:    result = (value << 1) + 6;
9:    printf("result = %d\n",result);
10:}
:
```

8.3.5 The G command: commence execution

The **g** command is used to commence execution of code on the target system. If no expression is supplied after the `g` command, execution will commence from the current value of PC (the program counter). If an expression is supplied, execution will commence from the address given. Execution will continue until a breakpoint is reached, or the user interrupts with control-`C`. After a breakpoint has been reached, execution can be continued from the same place using the `g`, `s` and `t` commands.

8.3.6 The I command: toggle instruction trace mode

The `i` command is used to toggle instruction trace mode. If instruction trace mode is enabled, each instruction is displayed before it is executed while stepping by entire C lines with the `s` command. For example, with instruction trace disabled, step behaves like this:

```
: s
result = 20
Stepped to
10:}
:
```

With instruction trace enabled, step will instead behave like this:

```
: s
_memtest+30H      push  r4
_memtest+32H      mov   r0,#04A2H
_memtest+36H      push  r0
_memtest+38H      fcall _printf
_memtest+3CH      adds  r7,#4
result = 20
Stepped to
10:}
:
```

Note that the library function *printf()* was not traced and thus operated properly and at full speed.

8.3.7 The L command: load a hex file

The `l` command is used to load object files into the target system. Lucifer correctly handles Motorola S-record format object files, Intel HEX files and binary images.

8.3.8 The M command: modify memory

The `m` command is used to write one or more values or ascii strings into memory at a specified address. This command takes the form:

```
m addr val1 [val2 ...]
```

where *addr* is the address to write to and all following arguments are values or strings to write to memory. Strings may use either single or double quotes. For example:

```
: m buf "hello" 13 10 'world' 0
```

8.3.9 The Q command: exit to operating system

The `q` command is used to exit from Lucifer to the operating system. Note: the `q` command does not stop the target system (that is, the Lucifer monitor running on the target system), so it is possible to re-enter Lucifer without re-initializing the target.

8.3.10 The R command: remove breakpoints

The r command is used to remove breakpoints which have been set with the b command. If no arguments are given the user is prompted for each breakpoint in turn. For example:

```
: r
Remove _main+$28 ? y
Remove _main+$44 ? n
Remove _test ? n
: r main+$44
Removed breakpoint _main+$44
:
```

8.3.11 The S command: step one line

The s command is used to step by one line of C or assembler code. For example:

```
: s
Stepped to
7:    scanf("%d", &value);
: s
Target wants input: 7
Stepped to
8:    result = (value << 1) + 6;
: s
Stepped to
9:    printf("result = %d\n",result);
: s
result = 20
Stepped to
10:}
:
```

This is normally implemented by executing several machine instruction single steps, and therefore can be quite slow. If Lucifer can determine that there are no function calls or control structures (*break*, *continue*, etc.) in the line, it will set a temporary breakpoint on the next line and execute the line at full speed. When single stepping by machine instructions, the step command will execute subroutine calls to external and library functions at full speed. This avoids the slow process of single stepping through complex library routines like *printf()*. Normal library console I/O works correctly during single stepping using the s command. Where no line number information is available, such as inside library routines, the s command becomes an assembler step like the t command.

8.3.12 The T command: trace one instruction

The `t` command is used to trace one machine instruction on the target. The current value of PC (the program counter) is used as the address of the instruction to be executed. After the instruction has been executed, the *next* instruction and the contents of all registers will be displayed.

8.3.13 The U command: disassemble machine instructions

The `u` command disassembles object code from the target system's memory. For example:

```
: u
9: printf("result = %d\n", result);
_memtest+30H      push  r4,r5
_memtest+32H      mov   r0,#04A2H
_memtest+36H      push  r0
_memtest+38H      fcall _printf
_memtest+3CH      adds  r7,#6
```

If an expression is supplied, the disassembly commences from the address supplied. If an address is not supplied, the disassembly commences from the instruction where the last disassembly ended. The disassembler automatically converts addresses in the object code to symbols if the symbol table for the program being disassembled is available. If the source code for a C program being disassembled is available, the C lines corresponding to each group of instructions are also displayed. Note: by default, any values specified will be interpreted as absolute addresses. If you want to specify a line number, prefix it with a colon.

8.3.14 The W command: upload binary

The `w` command is used to upload and write a chunk of target memory as a binary file. This command takes three arguments: filename, start address and length. The start address and length values are in hex. For example, if the Lucifer monitor ROM were at $7000 to $7FFF in the target system, it could be uploaded to a binary file with the command:

```
: w lucrom.bin 7000 1000
........
Uploaded 4096 (0x1000) bytes to lucrom.bin
:
```

8.3.15 The X command: examine or change registers

The `x` command is used to examine and change the contents of the target CPU registers. If no parameters are given, the registers are displayed without change. To change the contents of a register, two parameters must be supplied, a valid register name and the new value of the register. After setting a new register value, the contents of the registers are displayed. For example:

```
: x r1 1234 r0 3f
```

Any valid XA register name may be used. The PSW may be accessed as PSW for the entire 16 bit value, or as PSWL and PSWH for the low and high bytes respectively.

8.3.16 The @ command: display C variables

The @ command is used to examine the contents of memory interpreted as one of the standard C types. The form of the @ command is:

```
@t/[*]expr
```

where *t* is the type of the variable to be displayed, * consists of zero or more indirection operators ("*" or "*n**"), and *expr* is the address of the variable to be displayed. Table 8 - 3 on page 214, shows the available @ command variants.

For example, to display a long variable *longvar* in hex:

```
@lx longvar
```

To display a character, pointed to by a pointer *cptr*:

```
@c *cptr
```

To de-reference *ihandle*: a pointer to a pointer to an unsigned int:

```
@iu **ihandle
```

After displaying the variable, the current address is advanced by the size of the type displayed. This, makes it possible to step through arrays by repeatedly pressing return. On-line help for the @ command may be obtained by entering **?@** at the "**:**" prompt.

8.3.17 The . command: set a breakpoint and go

The **.** command is used to set a temporary breakpoint and resume execution from the current value of PC (the program counter). Execution continues until any breakpoint is reached or the user interrupts with control-C, then the temporary breakpoint is removed. Note: the temporary breakpoint is removed even if execution stops at a different breakpoint or is interrupted. If no breakpoint address is specified, the **.** command will display a list of active breakpoints.

```
: . 10
Target wants input: 7
result = 20
Breakpoint
```

Table 8 - 3 Lucifer @ command variants

Command	Type	Displays
@c	char	character and value
@cu	unsigned char	decimal
@cx	unsigned char	hexadecimal
@co	unsigned char	octal
@i	int	decimal
@iu	unsigned int	decimal
@ix	unsigned int	hexadecimal
@io	unsigned int	octal
@l	long	decimal
@lu	unsigned long	decimal
@lx	unsigned long	hexadecimal
@lo	unsigned long	octal
@f	float	decimal
@np	near pointer	symbol+offset
@p	pointer	symbol+offset
@s	string	string chars

```
10:}
main+$28        RET
:
```

8.3.18 The ; command: display from a source line

The `;` command is used to display 10 lines of source code from a specified position in a source file. If the line number is omitted, the last page of source code displayed will be re-displayed. For example:

```
: ; 4
4:
5: main()
6: {
7:    scanf("%d",&value);
8:    result = (value << 1) + 6;
9:    printf("result = %d\n",result);
10:}
```

8.3.19 The = command: display next page of source

The = command is used to display the next 10 lines of source code from the current file. For example, if the last source line displayed was line 7, = will display 10 lines starting from line 8.

8.3.20 The - command: display previous page of source

The - command is used to display the previous 10 lines of source code from the current file. For example, if the last page displayed started at line 15, - will display 10 lines starting from line 5.

8.3.21 The / command: search source file for a string

The / command is used to search the current source file for occurrences of a sequence of characters. Any text typed after the / is used to search the source file. The first source line containing the string specified is displayed. If no text is typed after the /, the previous search string will be used. Each string search starts from the point where the previous one finished, allowing the user to step through a source file finding all occurrences of a string.

```
: /printf
10:   printf("Enter a number:");
: /
14:   printf("Result = %d\n",answer);
: /
Can't find printf
:
```

8.3.22 The ! command: execute a DOS command

The **!** command is used to execute an operating system shell command line without exiting from Lucifer. Any text typed after the **!** is passed through to the shell without modification.

8.3.23 Other commands

In addition to the commands listed above, Lucifer will interpret any valid decimal number typed as a source line number and attempt to display the C source code for that line.

Pressing return without entering a command will result in re-execution of the previous command. In most cases the command resumes where the previous one left off. For example, if the previous command was *d 2000*, pressing return will have the same effect as the command *d 2010*.

If return is pressed after a breakpoint or **.** command has executed, it is equivalent to disassembling from the breakpoint address.

8.4 User Input and Output with Lucifer

The standard versions of the console I/O routines *putch()*, *getch()*, *getche()* and *init_uart()* are configured to work automatically with LUCIFER. Code which is downloaded under LUCIFER may use the standard I/O routines like *printf()* without any library modifications. Once you have finished debugging your code, you will need to insert into the library console, I/O routines suitable for your hardware. You can use the file **sources/getch.c** as a starting point.

8.5 Installing Lucifer on a Target

In order to use Lucifer on your target system, you will need to compile the Lucifer monitor and place it in a ROM. If your XA system already has a monitor in ROM, it is also possible to download the Lucifer target code into RAM. In most cases you will be able to use the Lucifer monitor program supplied without much modification.

Normally the only changes required will be the baud rate initialization in *init_uart()*. This requires modification for different baud rates or different clock frequencies. There are self-explanatory comments, in the code, at that point.

If you do not wish to use one of the XA internal serial ports you will need to modify the target code to access a different serial port.

8.5.1 Modifying the Target Code

Most modifications to *target.c* will be made to the serial I/O functions, *putch()*, *getch()* and *init_uart()*. For Lucifer to work correctly, you will need to have a system with common *code* and *external RAM* space as it is not possible to write to code memory on the XA. This can be achieved by mapping 32K of ROM from 0000h to 7FFFh and 32K of static RAM from 8000h to FFFFh and using PSEN and RD NOR'ed together for chip select.

8.5.2 Memory Mapping

Since the XA has separate, overlapping address spaces for code and data, special memory mapping is required to allow code to be downloaded and executed. It is important to map RAM into the first 64K of the data space, since the stack pointer in system mode is only 16 bits. Therefore a suggested arrangement is as follows. A ROM (size not important) is mapped into code space at 0. This will hold the Lucifer target program. A RAM of adequate size (say 128K bytes for example purposes) is mapped into data space at two locations; at 0, and at a higher address, say 80000h. It is also mapped into code space at the higher address, i.e. at 80000h it appears as both code and data.

This now allows Lucifer to download code into the ROM at the high address, and execute it from there. A portion of the RAM is reserved for data. The target code as supplied reserves 32K bytes of RAM for data, so user programs are downloaded at 88000h. The monitor program uses RAM from 7F00h to

7FFFh, so RAM from 20h (above the registers) to 7EFFh is available for the user program. With this arrangement, you would compile the Lucifer target program with the following addresses:

ROM address	0
RAM address	7F00
RAM size	100

and when compiling your application program you will compile with these addresses:

ROM address	88000
RAM address	20
RAM size	7EE0

8.5.3 Interrupts

Target.c reflects all unused interrupt vectors to the start of the user program area (88000 in the example above). This address is selected by the macro **RAMBASE** in *target.c*. All interrupts will be reflected to **RAMBASE**+vector, for example interrupt vector 84h will be reflected to 88084h. This means you do not have to change your program whether it is compiled to run under Lucifer or stand-alone in ROM, since the vector addresses are interpreted as being relative to the ROM start address.

Error messages

This chapter lists all possible error messages from the XA C compiler, with an explanation of each one.

'.' expected after '..' ***(Parser)***
The only context in which two successive dots may appear is as part of the ellipsis symbol, which must have 3 dots.

'case' not in switch ***(Parser)***
A case statement has been encountered but there is no enclosing switch statement. A case statement may only appear inside the body of a switch statement.

'default' not in switch ***(Parser)***
A label has been encountered called "default" but it is not enclosed by a switch statement. The label "default" is only legal inside the body of a switch statement.

(expected ***(Parser)***
An opening parenthesis was expected here. This must be the first token after a while, for, if, do or asm keyword.

) expected ***(Parser)***
A closing parenthesis was expected here. This may indicate you have left out a parenthesis in an expression, or you have some other syntax error.

: no match** ***(Preprocessor, Parser)
This is an internal compiler error. Contact HI-TECH Software technical support with details.

, expected ***(Parser)***
A comma was expected here. This probably means you have left out the comma between two identifiers in a declaration list. It may also mean that the immediately preceding type name is misspelled, and has thus been interpreted as an identifier.

-s, too few values specified in * ***(Preprocessor)***
The list of values to the preprocessor -S option is incomplete. This should not happen if the preprocessor is being invoked by the compiler driver or HPD.

-s, too many values, * unused ***(Preprocessor)***
There were too many values supplied to a -S preprocessor option.

... illegal in non-prototype arg list ***(Parser)***
The ellipsis symbol may only appear as the last item in a prototyped argument list. It may not appear on its own, nor may it appear after argument names that do not have types.

: expected ***(Parser)***
A colon is missing in a case label, or after the keyword "default". This often occurs when a semicolon is accidentally typed instead of a colon.

; expected ***(Parser)***
A semicolon is missing here. The semicolon is used as a terminator in many kinds of statements, e.g. do .. while, return etc.

= expected ***(Code Generator, Assembler)***
An equal sign was expected here.

#define syntax error ***(Preprocessor)***
A macro definition has a syntax error. This could be due to a macro or formal parameter name that does not start with a letter or a missing closing parenthesis (')').

#elif may not follow #else ***(Preprocessor)***
If a #else has been used after #if, you cannot then use a #elif in the same conditional block.

#elif must be in an #if ***(Preprocessor)***
#elif must be preceded by a matching #if line. If there is an apparently corresponding #if line, check for things like extra #endif's, or improperly terminated comments.

#else may not follow #else ***(Preprocessor)***
There can be only one #else corresponding to each #if.

#else must be in an #if ***(Preprocessor)***
#else can only be used after a matching #if.

#endif must be in an #if ***(Preprocessor)***
There must be a matching #if for each #endif. Check for the correct number of #ifs.

#error: * ***(Preprocessor)***
This is a programmer generated error; there is a directive causing a deliberate error. This is normally used to check compile time defines etc.

#if ... sizeof() syntax error ***(Preprocessor)***
The preprocessor found a syntax error in the argument to sizeof, in a #if expression. Probable causes are mismatched parentheses and similar things.

#if ... sizeof: bug, unknown type code * ***(Preprocessor)***

The preprocessor has made an internal error in evaluating a sizeof() expression. Check for a malformed type specifier.

#if ... sizeof: illegal type combination ***(Preprocessor)***

The preprocessor found an illegal type combination in the argument to sizeof() in a #if expression. Illegal combinations include such things as "short long int".

#if bug, operand = * ***(Preprocessor)***

The preprocessor has tried to evaluate an expression with an operator it does not understand. This is an internal error.

#if sizeof() error, no type specified ***(Preprocessor)***

Sizeof() was used in a preprocessor #if expression, but no type was specified. The argument to sizeof() in a preprocessor expression must be a valid simple type, or pointer to a simple type.

#if sizeof, unknown type * ***(Preprocessor)***

An unknown type was used in a preprocessor sizeof(). The preprocessor can only evaluate sizeof() with basic types, or pointers to basic types.

#if value stack overflow ***(Preprocessor)***

The preprocessor filled up its expression evaluation stack in a #if expression. Simplify the expression - it probably contains too many parenthesized subexpressions.

#if, #ifdef, or #ifndef without an argument ***(Preprocessor)***

The preprocessor directives #if, #ifdef and #ifndef must have an argument. The argument to #if should be an expression, while the argument to #ifdef or #ifndef should be a single name.

#include syntax error ***(Preprocessor)***

The syntax of the filename argument to #include is invalid. The argument to #include must be a valid file name, either enclosed in double quotes ("") or angle brackets (< >). For example:

```
#include "afile.h"
#include <otherfile.h>
```

Spaces should not be included, and the closing quote or bracket must be present. There should be nothing else on the line.

#included file * was converted to lower case ***(Preprocessor)***

The #include file name had to be converted to lowercase before it could be opened.

] expected ***(Parser)***

A closing square bracket was expected in an array declaration or an expression using an array index.

{ expected *(Parser)*
An opening brace was expected here.

} expected *(Parser)*
A closing brace was expected here.

a parameter may not be a function *(Parser)*
A function parameter may not be a function. It may be a pointer to a function, so perhaps a "*" has been omitted from the declaration.

absolute expression required *(Assembler)*
An absolute expression is required in this context.

add_reloc - bad size *(Assembler)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

ambiguous format name '*' *(Cromwell)*
The output format specified to Cromwell is ambiguous.

argument * conflicts with prototype *(Parser)*
The argument specified (argument 1 is the left most argument) of this function declaration does not agree with a previous prototype for this function.

argument -w* ignored *(Linker)*
The argument to the linker option -w is out of range. For warning levels, the range is -9 to 9. For the map file width, the range is greater than or equal to 10.

argument list conflicts with prototype *(Parser)*
The argument list in a function definition is not the same as a previous prototype for that function. Check that the number and types of the arguments are all the same.

argument redeclared: * *(Parser)*
The specified argument is declared more than once in the same argument list.

argument too long *(Preprocessor, Parser)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

arithmetic overflow in constant expression *(Code Generator)*
A constant expression has been evaluated by the code generator that has resulted in a value that is too big for the type of the expression, e.g. trying to store the value 256 in a "char".

array dimension on * ignored *(Preprocessor)*
An array dimension on a function parameter has been ignored because the argument is actually converted to a pointer when passed. Thus arrays of any size may be passed.

array dimension redeclared *(Parser)*
An array dimension has been declared as a different non-zero value from its previous declaration. It is acceptable to redeclare the size of an array that was previously declared with a zero dimension, but not otherwise.

array index out of bounds *(Parser)*
An array is being indexed with a constant value that is less than zero, or greater than or equal to the number of elements in the array.

assertion *(Code Generator)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

assertion failed: * *(Linker)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

attempt to modify const object *(Parser)*
Objects declared "const" may not be assigned to or modified in any other way.

auto variable * should not be qualified *(Parser)*
An auto variable should not have qualifiers such as "near" or "far" associated with it. Its storage class is implicitly defined by the stack organization.

bad #if ... defined() syntax *(Preprocessor)*
The defined() pseudo-function in a preprocessor expression requires its argument to be a single name. The name must start with a letter. It should be enclosed in parentheses.

bad '-p' format *(Linker)*
The "-P" option given to the linker is malformed.

bad -a spec: * *(Linker)*
The format of a -A specification, giving address ranges to the linker, is invalid. The correct format is:

-Aclass=low-high

where class is the name of a psect class, and low and high are hex numbers.

bad -m option: * *(Code Generator)*
The code generator has been passed a -M option that it does not understand. This should not happen if it is being invoked by a standard compiler driver.

bad -q option * *(Parser)*
The first pass of the compiler has been invoked with a -Q option, to specify a type qualifier name, that is badly formed.

bad arg * to tysize *(Parser)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

bad arg to e: * *(Code Generator)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

bad bconfloat - * *(Code Generator)*
This is an internal code generator error. Contact HI-TECH technical support with full details of the code that caused this error.

bad bit address *(Assembler, Optimiser)*
The address supplied is not a bit-addressable portion of the XA. Bit addressable portions include the registers R0 to R15, direct RAM from 20h to 3Fh, and the on-chip SFRs from 400h to 43Fh.

bad bit expression *(Optimiser)*
There is a bad bit expression in the assembler file.

bad bit number *(Assembler, Optimiser)*
A bit number must be an absolute expression in the range 0-7.

bad bitfield type *(Parser)*
A bitfield may only have a type of int.

bad character const *(Parser, Assembler, Optimiser)*
This character constant is badly formed.

bad character in extended tekhex line * *(Objtohex)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

bad checksum specification *(Linker)*
A checksum list supplied to the linker is syntatically incorrect.

bad combination of flags *(Objtohex)*
The combination of options supplied to objtohex is invalid.

bad complex range check *(Linker)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

bad complex relocation *(Linker)*
The linker has been asked to perform complex relocation that is not syntactically correct. Probably means a corrupted object file.

bad confloat - * *(Code Generator)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

bad conval - * *(Code Generator)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

bad dimensions *(Code Generator)*
The code generator has been passed a declaration that results in an array having a zero dimension.

bad dp/nargs in openpar: c = * *(Preprocessor)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

bad element count expr *(Code Generator)*
There is an error in the intermediate code. Try re-installing the compiler from the distribution disks, as this could be caused by a corrupted file.

bad fixup value *(Optimiser)*
The assembler file passed to the optimizer is invalid.

bad gn *(Code Generator)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

bad high address in -a spec *(Linker)*
The high address given in a -A specification is invalid: it should be a valid number, in decimal, octal or hexadecimal radix. The radix is specified by a trailing O (for octal) or H for hex. Decimal is default.

bad int. code *(Code Generator)*
The code generator has been passed input that is not syntatically correct.

bad load address in -a spec *(Linker)*
The load address given in a -A specification is invalid: it should be a valid number, in decimal, octal or hexadecimal radix. The radix is specified by a trailing O (for octal) or H for hex. Decimal is default.

bad low address in -a spec *(Linker)*
The low address given in a -A specification is invalid: it should be a valid number, in decimal, octal or hexadecimal radix. The radix is specified by a trailing O (for octal) or H for hex. Decimal is default.

bad min (+) format in spec *(Linker)*
The minimum address specification in the linker's -p option is badly formatted.

bad mod '+' for how = * *(Code Generator)*
Internal error - Contact HI-TECH.

bad non-zero node in call graph *(Linker)*
The linker has encountered a top level node in the call graph that is referenced from lower down in the call graph. This probably means the program has indirect recursion, which is not allowed when using a compiled stack.

bad object code format *(Linker)*
The object code format of this object file is invalid. This probably means it is either truncated, corrupted, or not a HI-TECH object file.

bad op * to revlog *(Code Generator)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

bad op * to swaplog *(Code Generator)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

bad op: "*" *(Code Generator)*
This is caused by an error in the intermediate code file. You may have run out of disk space for temporary files.

bad operand *(Optimiser)*
This operand is invalid. Check the syntax.

bad operand to seg *(Assembler, Optimiser)*
This can happen if you try to take the segment part of something that is already a segment address.

bad origin format in spec *(Linker)*
The origin format in a -p option is not a validly formed decimal, octal or hex number. A hex number must have a trailing H.

bad overrun address in -a spec *(Linker)*
The overrun address given in a -A specification is invalid: it should be a valid number, in decimal, octal or hexadecimal radix. The radix is specified by a trailing O (for octal) or H for hex. Decimal is default.

bad popreg: * *(Code Generator)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

bad pragma * *(Code Generator)*
The code generator has been passed a "pragma" directive that it does not understand.

bad pushreg: * *(Code Generator)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

bad putwsize *(Code Generator)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

bad record type * *(Linker)*
This indicates that the object file is not a valid HI-TECH object file.

bad register for structret
This is an internal compiler error. Contact HI-TECH Software technical support with details.

bad relocation type *(Assembler)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

bad repeat count in -a spec *(Linker)*
The repeat count given in a -A specification is invalid: it should be a valid decimal number.

bad ret_mask *(Code Generator)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

bad segment fixups *(Objtohex)*
This is an obscure message from objtohex that is not likely to occur in practice.

bad segspec * *(Linker)*
The segspec option (-G) to the linker is invalid. The correct form of a segspec option is along the following lines:

-Gnxc+o

where n stands for the segment number, x is a multiplier symbol, c is a constant (multiplier) and o is a constant offset. For example the option

-Gnx4+16

would assign segment selectors starting from 16, and incrementing by 4 for each segment, i.e. in the order 16, 20, 24 etc.

bad size in -s option *(Linker)*
The size part of a -S option is not a validly formed number. The number must be a decimal, octal or hex number. A hex number needs a trailing H, and an octal number a trailing O. All others are assumed to be decimal.

bad size in index_type *(Parser)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

bad size list *(Parser)*
The first pass of the compiler has been invoked with a -Z option, specifying sizes of types, that is badly formed.

bad storage class *(Code Generator)*
The storage class "auto" may only be used inside a function. A function parameter may not have any storage class specifier other than "register". If this error is issued by the code generator, it could mean that the intermediate code file is invalid. This could be caused by running out of disk space.

bad string * in psect pragma ***(Code Generator)***
The code generator has been passed a "pragma psect" directive that has a badly formed string. "Pragma psect" should be followed by something of the form "oldname=newname".

bad switch size * ***(Code Generator)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

bad sx ***(Code Generator)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

bad token in register list ***(Optimiser)***
The register list must include only registers.

bad u usage ***(Code Generator)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

bad variable syntax ***(Code Generator)***
There is an error in the intermediate code file. This could be caused by running out of disk space for temporary files.

bad which * after i ***(Code Generator)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

binary digit expected ***(Parser)***
A binary digit was expected. The format for a binary number is 0Bxxx where xxx is a string containing zeroes and/or ones, e.g.

0B0110

bit address overflow ***(Assembler)***
The bit address supplied is outside the XA bit space. The bit address should be in the bit space which has a range 0h to 3FFh.

bit field too large (* bits) ***(Code Generator)***
The maximum number of bits in a bit field is the same as the number of bits in an "int".

bit number not absolute ***(Optimiser)***
A bit number must be an absolute number in the range 0-7.

bit range check failed * ***(Linker)***
The bit addressing was out of range.

bit variables must be global or static ***(Code Generator)***
A bit variable cannot be of type auto. If you require a bit variable with scope local to a block of code or function, qualify it static.

bitfield comparison out of range ***(Code Generator)***
This is the result of comparing a bitfield with a value when the value is out of range of the bitfield. For example, comparing a 2-bit bitfield to the value 5 will never be true as a 2-bit bitfield has a range from 0 to 3,

bug: illegal __ macro * ***(Preprocessor)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

call depth exceeded by * ***(Linker)***
The call graph shows that functions are nested to a depth greater than specified.

can't allocate memory for arguments ***(Preprocessor, Parser)***
The compiler could not allocate any more memory. Try increasing the size of available memory.

can't be both far and near ***(Parser)***
It is illegal to qualify a type as both far and near.

can't be long ***(Parser)***
Only "int" and "float" can be qualified with "long". Thus combinations like "long char" are illegal.

can't be register ***(Parser)***
Only function parameters or auto (local) variables may be declared "register".

can't be short ***(Parser)***
Only "int" can be modified with short. Thus combinations like "short float" are illegal.

can't be unsigned ***(Parser)***
There is no such thing as an unsigned floating point number.

can't call an interrupt function ***(Parser)***
A function qualified "interrupt" can't be called from other functions. It can only be called by a hardware (or software) interrupt. This is because an interrupt function has special function entry and exit code that is appropriate only for calling from an interrupt. An "interrupt" function can call other non-interrupt functions.

can't create * ***(Code Generator, Assembler, Linker, Optimiser)***
The named file could not be created. Check that all directories in the path are present.

can't create cross reference file * ***(Assembler)***
The cross reference file could not be created. Check that all directories are present. This can also be caused by the assembler running out of memory.

can’t create temp file ***(Linker)***
The compiler was unable to create a temporary file. Check the DOS Environment variable TEMP (and TMP) and verify it points to a directory that exists, and that there is space available on that drive. For example, AUTOEXEC.BAT should have something like:

SET TEMP=C:\TEMP

where the directory C:\TEMP exists.

can’t create temp file * ***(Code Generator)***
The compiler could not create the temporary file named. Check that all the directories in the file path exist.

can’t enter abs psect ***(Assembler)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

can’t find op ***(Assembler, Optimiser)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

can’t find space for psect * in segment * ***(Linker)***
The named psect cannot be placed in the specified segment. This either means that the memory associated with the segment has been filled, or that the psect cannot be positioned in any of the available gaps in the memory. Split large functions (for CODE segements) in several smaller functions and ensure that the optimizers are being used.

can’t generate code for this expression ***(Code Generator)***
This expression is too difficult for the code generator to handle. Try simplifying the expression, e.g. using a temporary variable to hold an intermediate result.

can’t have ’signed’ and ’unsigned’ together ***(Parser)***
The type modifiers signed and unsigned cannot be used together in the same declaration, as they have opposite meaning.

can’t have an array of bits or a pointer to bit ***(Parser)***
It is not legal to have an array of bits, or a pointer to bit.

can’t have array of functions ***(Parser)***
You can’t have an array of functions. You can however have an array of pointers to functions. The correct syntax for an array of pointers to functions is "int (* arrayname[])();". Note that parentheses are used to associate the star (*) with the array name before the parentheses denoting a function.

can’t have arrays of bits ***(Code Generator)***
You can’t have an array of bits, because bits can’t be indexed.

can't have pointer to bit *(Code Generator)*
Bit variables as implemented in the 8051 compiler are not addressable via pointers, so a pointer to a bit is not allowed.

can't initialize arg *(Parser)*
A function argument can't have an initialiser. The initialisation of the argument happens when the function is called and a value is provided for the argument by the calling function.

can't initialize bit type *(Code Generator)*
Variables of type bit cannot be initialised.

can't mix proto and non-proto args *(Parser)*
A function declaration can only have all prototyped arguments (i.e. with types inside the parentheses) or all K&R style args (i.e. only names inside the parentheses and the argument types in a declaration list before the start of the function body).

can't open *(Linker)*
A file can't be opened - check spelling.

can't open * *(Code Generator, Assembler, Optimiser, Cromwell)*
The named file could not be opened. Check the spelling and the directory path. This can also be caused by running out of memory.

can't open avmap file * *(Linker)*
A file required for producing Avocet format symbol files is missing. Try re-installing the compiler.

can't open checksum file * *(Linker)*
The checksum file specified to objtohex could not be opened. Check spelling etc.

can't open command file * *(Preprocessor, Linker)*
The command file specified could not be opened for reading. Check spelling!

can't open error file * *(Linker)*
The error file specified using the -e option could not be opened.

can't open include file * *(Assembler)*
The named include file could not be opened. Check spelling. This can also be caused by running out of memory, or running out of file handles.

can't open input file * *(Preprocessor, Assembler)*
The specified input file could not be opened. Check the spelling of the file name.

can't open output file * *(Preprocessor, Assembler)*
The specified output file could not be created. This could be because a directory in the path name does not exist.

can’t reopen * ***(Parser)***
The compiler could not reopen a temporary file it had just created.

can’t seek in * ***(Linker)***
The linker can’t seek in the specified file. Make sure the output file is a valid filename.

can’t take address of register variable ***(Parser)***
A variable declared "register" may not have storage allocated for it in memory, and thus it is illegal to attempt to take the address of it by applying the "&" operator.

can’t take sizeof func ***(Parser)***
Functions don’t have sizes, so you can’t take use the "sizeof" operator on a function.

can’t take sizeof(bit) ***(Parser)***
You can’t take sizeof a bit value, since it is smaller than a byte.

can’t take this address ***(Parser)***
The expression which was the object of the "&" operator is not one that denotes memory storage ("an lvalue") and therefore its address can not be defined.

can’t use a string in an #if ***(Preprocessor)***
The preprocessor does not allow the use of strings in #if expressions.

cannot get memory ***(Linker)***
The linker is out of memory! This is unlikely to happen, but removing TSR’s etc. is the cure.

cannot open ***(Linker)***
A file cannot be opened - check spelling.

cannot open include file * ***(Preprocessor)***
The named include file could not be opened for reading by the preprocessor. Check the spelling of the filename. If it is a standard header file, not in the current directory, then the name should be enclosed in angle brackets (<>) not quotes.

cast type must be scalar or void ***(Parser)***
A typecast (an abstract type declarator enclosed in parentheses) must denote a type which is either scalar (i.e. not an array or a structure) or the type "void".

char const too long ***(Parser)***
A character constant enclosed in single quotes may not contain more than one character.

character not valid at this point in format specifier ***(Parser)***
The printf() style format specifier has an illegal character.

checksum error in intel hex file *, line * *(Cromwell)*
A checksum error was found at the specified line in the specified Intel hex file. The file may have been corrupted.

circular indirect definition of symbol * *(Linker)*
The specified symbol has been equated to an external symbol which, in turn, has been equated to the first symbol.

class * memory space redefined: */* *(Linker)*
A class has been defined in two different memory spaces. Either rename one of the classes or, if they are the same class, place them in the same memory space.

close error (disk space?) *(Parser)*
When the compiler closed a temporary file, an error was reported. The most likely cause of this is that there was insufficient space on disk for the file.

common symbol psect conflict: * *(Linker)*
A common symbol has been defined to be in more than one psect.

complex relocation not supported for -r or -l options yet *(Linker)*
The linker was given a -R or -L option with file that contain complex relocation. This is not yet supported.

conflicting fnconf records *(Linker)*
This is probably caused by multiple run-time startoff module. Check the linker arguments, or "Object Files..." in HPD.

constant conditional branch *(Code Generator)*
A conditional branch (generated by an "if" statement etc.) always follows the same path. This may indicate an expression with missing or badly placed parentheses, causing the evaluation to yield a value different to what you expected, or it may be because you have written something like "while(1)". To produce an infinite loop, use "for(;;)".

constant conditional branch: possible use of = instead of == *(Code Generator)*
There is an expression inside an if or other conditional construct, where a constant is being assigned to a variable. This may mean you have inadvertently used an assignment (=) instead of a compare (==).

constant expression required *(Parser)*
In this context an expression is required that can be evaluated to a constant at compile time.

constant left operand to ? *(Code Generator)*
The left operand to a conditional operator (?) is constant, thus the result of the tertiary operator ?: will always be the same.

constant operand to || or && ***(Code Generator)***
One operand to the logical operators || or && is a constant. Check the expression for missing or badly placed parentheses.

constant relational expression ***(Code Generator)***
There is a relational expression that will always be true or false. This may be because e.g. you are comparing an unsigned number with a negative value, or comparing a variable with a value greater than the largest number it can represent.

control line * within macro expansion ***(Preprocessor)***
A preprocessor control line (one starting with a #) has been encountered while expanding a macro. This should not happen.

conversion to shorter data type ***(Code Generator)***
Truncation may occur in this expression as the lvalue is of shorter type than the rvalue.

copyexpr: can't handle v_rtype = * ***(Assembler)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

declaration of * hides outer declaration ***(Parser)***
An object has been declared that has the same name as an outer declaration (i.e. one outside and preceding the current function or block). This is legal, but can lead to accidental use of one variable when the outer one was intended.

declarator too complex ***(Parser)***
This declarator is too complex for the compiler to handle. Examine the declaration and find a way to simplify it. If the compiler finds it too complex, so will anybody maintaining the code.

default case redefined ***(Parser)***
There is only allowed to be one "default" label in a switch statement. You have more than one.

def[bmsf] in text psect ***(Optimiser)***
The assembler file supplied to the optimizer is invalid.

degenerate signed comparison ***(Code Generator)***
There is a comparision of a signed value with the most negative value possible for this type, such that the comparision will always be true or false. E.g. char c;

if(c >= -128)

will always be true, because an 8 bit signed char has a maximum negative value of -128.

degenerate unsigned comparison ***(Code Generator)***
There is a comparision of an unsigned value with zero, which will always be true or false. E.g.

unsigned char c;
if(c >= 0)

will always be true, because an unsigned value can never be less than zero.

delete what ? ***(Libr)***
The librarian requires one or more modules to be listed for deletion when using the 'd' key.

did not recognize format of input file ***(Cromwell)***
The input file to Cromwell is required to be COD, Intel HEX, Motorola HEX, COFF, OMF51, P&E or HI-TECH.

digit out of range ***(Parser, Assembler, Optimiser)***
A digit in this number is out of range of the radix for the number, e.g. using the digit 8 in an octal number, or hex digits A-F in a decimal number. An octal number is denoted by the digit string commencing with a zero, while a hex number starts with "0X" or "0x".

dimension required ***(Parser)***
Only the most significant (i.e. the first) dimension in a multi-dimension array may not be assigned a value. All succeeding dimensions must be present.

direct address overflow ***(Assembler)***
An attempt was made to directly access memory outside of the directly-addressed RAM space or the SFR space.

direct range check failed * ***(Linker)***
The direct addressing was out of range.

divide by zero in #if, zero result assumed ***(Preprocessor)***
Inside a #if expression, there is a division by zero which has been treated as yielding zero.

division by zero ***(Code Generator)***
A constant expression that was being evaluated involved a division by zero.

double float argument required ***(Parser)***
The printf format specifier corresponding to this argument is %f or similar, and requires a floating point expression. Check for missing or extra format specifiers or arguments to printf.

duplicate -d or -h flag ***(Linker)***
The a symbol file name has been specified to the linker for a second time.

duplicate -m flag ***(Linker)***
The linker only likes to see one -m flag, unless one of them does not specify a file name. Two map file names are more than it can handle!

duplicate case label * ***(Code Generator)***
There is more than one case label with this value in a switch statement.

duplicate fnconf directive ? ***(Assembler)***
.

duplicate label * ***(Parser)***
The same name is used for a label more than once in this function. Note that the scope of labels is the entire function, not just the block that encloses a label.

duplicate qualifier ***(Parser)***
There are two occurrences of the same qualifier in this type specification. This can occur either directly or through the use of a typedef. Remove the redundant qualifier.

duplicate qualifier key * ***(Parser)***
This qualifier key (given via a -Q option) has been used twice.

duplicate qualifier name * ***(Parser)***
A duplicate qualifier name has been specified to P1 via a -Q option. This should not occur if the standard compiler drivers are used.

duplicate register in rlist ***(Assembler)***
The specified register has already specified for pushing onto or popping from the stack.

end of file within macro argument from line * ***(Preprocessor)***
A macro argument has not been terminated. This probably means the closing parenthesis has been omitted from a macro invocation. The line number given is the line where the macro argument started.

end of string in format specifier ***(Parser)***
The format specifier for the printf() style function is malformed.

end statement inside include file or macro ***(Assembler)***
An END statement was found inside an include file or a macro.

entry point multiply defined ***(Linker)***
There is more than one entry point defined in the object files given the linker.

enum tag or { expected ***(Parser)***
After the keyword "enum" must come either an identifier that is or will be defined as an enum tag, or an opening brace.

eof in #asm ***(Preprocessor)***
An end of file has been encountered inside a #asm block. This probably means the #endasm is missing or misspelt.

eof in comment *(Preprocessor)*
End of file was encountered inside a comment. Check for a missing closing comment flag.

eof inside conditional *(Assembler)*
END-of-FILE was encountered while scanning for an "endif" to match a previous "if".

eof inside macro def'n *(Assembler)*
End-of-file was encountered while processing a macro definition. This means there is a missing "endm" directive.

eof on string file *(Parser)*
P1 has encountered an unexpected end-of-file while re-reading its file used to store constant strings before sorting and merging. This is most probably due to running out of disk space. Check free disk space.

errfile *(Assembler)*
The error file specified using the -Efile option could not be opened.

error closing output file *(Code Generator, Optimiser)*
The compiler detected an error when closing a file. This most probably means there is insufficient disk space.

error dumping * *(Cromwell)*
Either the input file to Cromwell is of an unsupported type or that file cannot be dumped to the screen.

error in format string *(Parser)*
There is an error in the format string here. The string has been interpreted as a printf() style format string, and it is not syntactically correct. If not corrected, this will cause unexpected behaviour at run time.

evaluation period has expired *(Driver)*
The evaluation period for this compiler has expired. Contact HI-TECH to purchase a full licence.

expand - bad how *(Code Generator)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

expand - bad which *(Code Generator)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

expected '-' in -a spec *(Linker)*
There should be a minus sign (-) between the high and low addresses in a -A spec, e.g.

-AROM=1000h-1FFFh

exponent expected *(Parser)*
A floating point constant must have at least one digit after the "e" or "E".

expression error ***(Code Generator, Assembler, Optimiser)***
There is a syntax error in this expression, OR there is an error in the intermediate code file. This could be caused by running out of disk space.

expression generates no code ***(Code Generator)***
This expression generates no code. Check for things like leaving off the parentheses in a function call.

expression stack overflow at op * ***(Preprocessor)***
Expressions in #if lines are evaluated using a stack with a size of 128. It is possible for very complex expressions to overflow this. Simplify the expression.

expression syntax ***(Parser)***
This expression is badly formed and cannot be parsed by the compiler.

expression too complex ***(Parser)***
This expression has caused overflow of the compiler's internal stack and should be re-arranged or split into two expressions.

external declaration inside function ***(Parser)***
A function contains an "extern" declaration. This is legal but is invariably A Bad Thing as it restricts the scope of the function declaration to the function body. This means that if the compiler encounters another declaration, use or definition of the extern object later in the same file, it will no longer have the earlier declaration and thus will be unable to check that the declarations are consistent. This can lead to strange behaviour of your program or signature errors at link time. It will also hide any previous declarations of the same thing, again subverting the compiler's type checking. As a general rule, always declare "extern" variables and functions outside any other functions.

field width not valid at this point ***(Parser)***
A field width may not appear at this point in a printf() type format specifier.

file name index out of range in line no. record ***(Cromwell)***
The .COD file has an invalid format in the specified record.

filename work buffer overflow ***(Preprocessor)***
A filename constructed while looking for an include file has exceeded the length of an internal buffer. Since this buffer is 4096 bytes long, this is unlikely to happen.

fixup overflow in expression * ***(Linker)***
The linker was asked to relocate (fixup) an item that would not fit back into the space after relocation. For example this will occur if a byte size object is initialized with an address that is bigger than 255. This error occurred in a complex expression.

fixup overflow referencing * *(Linker)*
The linker was asked to relocate (fixup) an item that would not fit back into the space after relocation. For example this will occur if a byte size object is initialized with an address that is bigger than 255.

flag * unknown *(Assembler)*
This option used on a "PSECT" directive is unknown to the assembler.

float param coerced to double *(Parser)*
Where a non-prototyped function has a parameter declared as "float", the compiler converts this into a "double float". This is because the default C type conversion conventions provide that when a floating point number is passed to a non-prototyped function, it will be converted to double. It is important that the function declaration be consistent with this convention.

floating number expected *(Assembler)*
The arguments to the "DEFF" pseudo-op must be valid floating point numbers.

form length must be >= 15 *(Assembler)*
The form length specified using the -Flength option must be at least 15 lines.

formal parameter expected after # *(Preprocessor)*
The stringization operator # (not to be confused with the leading # used for preprocessor control lines) must be followed by a formal macro parameter. If you need to stringize a token, you will need to define a special macro to do it, e.g.

#define __mkstr__(x) #x

then use __mkstr__(token) wherever you need to convert a token into a string.

function * appears in multiple call graphs: rooted at * *(Linker)*
This function can be called from both main line code and interrupt code. Use the reentrant keyword, if this compiler supports it, or recode to avoid using local variables or parameters, or duplicate the function.

function * argument evaluation overlapped *(Linker)*
A function call involves arguments which overlap between two functions. This could occur with a call like:

void fn1(void) { fn3(7, fn2(3), fn2(9)); /* Offending call */ } char fn2(char fred) { return fred + fn3(5,1,0); } char fn3(char one, char two, char three) { return one+two+three; }

where fn1 is calling fn3, and two arguments are evaluated by calling fn2, which in turn calls fn3. The structure should be modified to prevent this.

function * is never called ***(Linker)***

This function is never called. This may not represent a problem, but space could be saved by removing it. If you believe this function should be called, check your source code.

function body expected ***(Parser)***

Where a function declaration is encountered with K&R style arguments (i.e. argument names but no types inside the parentheses) a function body is expected to follow.

function declared implicit int ***(Parser)***

Where the compiler encounters a function call of a function whose name is presently undefined, the compiler will automatically declare the function to be of type "int", with unspecified (K&R style) parameters. If a definition of the function is subsequently encountered, it is possible that its type and arguments will be different from the earlier implicit declaration, causing a compiler error. The solution is to ensure that all functions are defined or at least declared before use, preferably with prototyped parameters. If it is necessary to make a forward declaration of a function, it should be preceded with the keywords "extern" or "static" as appropriate.

function does not take arguments ***(Parser, Code Generator)***

This function has no parameters, but it is called here with one or more arguments.

function is already 'extern'; can't be 'static' ***(Parser)***

This function was already declared extern, possibly through an implicit declaration. It has now been redeclared static, but this redeclaration is invalid. If the problem has arisen because of use before definition, either move the definition earlier in the file, or place a static forward definition earlier in the file, e.g. static int fred(void);

function or function pointer required ***(Parser)***

Only a function or function pointer can be the subject of a function call. This error can be produced when an expression has a syntax error resulting in a variable or expression being followed by an opening parenthesis ("(") which denotes a function call.

functions can't return arrays ***(Parser)***

A function can return only a scalar (simple) type or a structure. It cannot return an array.

functions can't return functions ***(Parser)***

A function cannot return a function. It can return a function pointer. A function returning a pointer to a function could be declared like this: int (* (name()))(). Note the many parentheses that are necessary to make the parts of the declaration bind correctly.

functions nested too deep ***(Code Generator)***

This error is unlikely to happen with C code, since C cannot have nested functions!

garbage after operands *(Assembler)*
There is something on this line after the operands other than a comment. This could indicate an operand error.

garbage on end of line *(Assembler)*
There were non-blank and non-comment characters after the end of the operands for this instruction. Note that a comment must be started with a semicolon.

hex digit expected *(Parser)*
After "0x" should follow at least one of the hex digits 0-9 and A-F or a-f.

I/O error reading symbol table *(Cromwell)*
Cromwell could not read the symbol table. This could be because the file was truncated or there was some other problem reading the file.

ident records do not match *(Linker)*
The object files passed to the linker do not have matching ident records. This means they are for different processor types.

identifier expected *(Parser)*
Inside the braces of an "enum" declaration should be a comma-separated list of identifiers.

identifier redefined: * *(Parser)*
This identifier has already been defined. It cannot be defined again.

identifier redefined: * (from line *) *(Parser)*
This identifier has been defined twice. The 'from line' value is the line number of the first declaration.

illegal # command * *(Preprocessor)*
The preprocessor has encountered a line starting with #, but which is not followed by a recognized control keyword. This probably means the keyword has been misspelt. Legal control keywords are: assert, asm, define, elif, else, endasm, endif, error, if, ifdef, ifndef, include, line, pragma, undef.

illegal #if line *(Preprocessor)*
There is a syntax error in the expression following #if. Check the expression to ensure it is properly constructed.

illegal #undef argument *(Preprocessor)*
The argument to #undef must be a valid name. It must start with a letter.

illegal '#' directive *(Preprocessor, Parser)*
The compiler does not understand the "#" directive. It is probably a misspelling of a pre-processor "#" directive.

illegal align bound ***(Assembler)***
An illegal align bound was specified in the psect ALIGN= directive. Possible values include 0 or 1 to align to the byte, 2 to align to the word, and 4 to align to the double word.

illegal bit number in byte register ***(Assembler, Optimiser)***
A bit number outside the range 0 to 7 has been used for a byte register.

illegal bit number in word register ***(Assembler, Optimiser)***
A bit number outside the range 0 to 15 has been used for a word register.

illegal branch target
The target of a branch is on an odd address. This is not allowed. Use the psect align directive to align labels to an even address.

illegal character (* decimal) in #if ***(Preprocessor)***
The #if expression had an illegal character. Check the line for correct syntax.

illegal character * ***(Parser)***
This character is illegal.

illegal character * in #if ***(Preprocessor)***
There is a character in a #if expression that has no business being there. Valid characters are the letters, digits and those comprising the acceptable operators.

illegal conversion ***(Parser)***
This expression implies a conversion between incompatible types, e.g. a conversion of a structure type into an integer.

illegal conversion between pointer types ***(Parser)***
A pointer of one type (i.e. pointing to a particular kind of object) has been converted into a pointer of a different type. This will usually mean you have used the wrong variable, but if this is genuinely what you want to do, use a typecast to inform the compiler that you want the conversion and the warning will be suppressed.

illegal conversion of integer to pointer ***(Parser)***
An integer has been assigned to or otherwise converted to a pointer type. This will usually mean you have used the wrong variable, but if this is genuinely what you want to do, use a typecast to inform the compiler that you want the conversion and the warning will be suppressed.

illegal conversion of pointer to integer ***(Parser)***
A pointer has been assigned to or otherwise converted to a integral type. This will usually mean you have used the wrong variable, but if this is genuinely what you want to do, use a typecast to inform the compiler that you want the conversion and the warning will be suppressed.

illegal flag * ***(Linker)***
This flag is unrecognized.

illegal function qualifier(s) ***(Parser)***
A qualifier such as "const" or "volatile" has been applied to a function. These qualifiers only make sense when used with an lvalue (i.e. an expression denoting memory storage). Perhaps you left out a star ("*") indicating that the function should return a pointer to a qualified object.

illegal index register ***(Optimiser)***
The optimiser has found a byte register being used for indexing instead of a word register.

illegal initialisation ***(Parser)***
You can’t initialise a "typedef" declaration, because it does not reserve any storage that could be initialised.

illegal operation on a bit variable ***(Parser)***
Not all operations on bit variables are supported. This operation is one of those.

illegal operator in #if ***(Preprocessor)***
A #if expression has an illegal operator. Check for correct syntax.

illegal or too many -g flags ***(Linker)***
There has been more than one -g option, or the -g option did not have any arguments following. The arguments specify how the segment addresses are calculated.

illegal or too many -o flags ***(Linker)***
This -o flag is illegal, or another -o option has been encountered. A -o option to the linker must have a filename. There should be no space between the filename and the -o, e.g. -ofile.obj

illegal or too many -p flags ***(Linker)***
There have been too many -p options passed to the linker, or a -p option was not followed by any arguments. The arguments of separate -p options may be combined and separated by commas.

illegal record type ***(Linker)***
There is an error in an object file. This is either an invalid object file, or an internal error in the linker. Try recreating the object file.

illegal relocation size: * ***(Linker)***
There is an error in the object code format read by the linker. This either means you are using a linker that is out of date, or that there is an internal error in the assembler or linker.

illegal relocation type: * ***(Linker)***
An object file contained a relocation record with an illegal relocation type. This probably means the file is corrupted or not an object file.

illegal switch * ***(Code Generator, Assembler, Optimiser)***
This command line option was not understood.

illegal type for array dimension ***(Parser)***
An array dimension must be either an integral type or an enumerated value.

illegal type for index expression ***(Parser)***
An index expression must be either integral or an enumerated value.

illegal type for switch expression ***(Parser)***
A "switch" operation must have an expression that is either an integral type or an enumerated value.

illegal use of void expression ***(Parser)***
A void expression has no value and therefore you can't use it anywhere an expression with a value is required, e.g. as an operand to an arithmetic operator.

image too big ***(Objtohex)***
The program image being constructed by objtohex is too big for its virtual memory system.

implicit conversion of float to integer ***(Parser)***
A floating point value has been assigned or otherwise converted to an integral type. This could result in truncation of the floating point value. A typecast will make this warning go away.

implicit return at end of non-void function ***(Parser)***
A function which has been declared to return a value has an execution path that will allow it to reach the end of the function body, thus returning without a value. Either insert a return statement with a value, or if the function is not to return a value, declare it "void".

implict signed to unsigned conversion ***(Parser)***
A signed number is being assigned or otherwise converted to a larger unsigned type. Under the ANSI "value preserving" rules, this will result in the signed value being first sign-extended to a signed number the size of the target type, then converted to unsigned (which involves no change in bit pattern). Thus an unexpected sign extension can occur. To ensure this does not happen, first convert the signed value to an unsigned equivalent, e.g. if you want to assign a signed char to an unsigned int, first typecast the char value to "unsigned char".

inappropriate 'else' ***(Parser)***
An "else" keyword has been encountered that cannot be associated with an "if" statement. This may mean there is a missing brace or other syntactic error.

inappropriate break/continue ***(Parser)***
A "break" or "continue" statement has been found that is not enclosed in an appropriate control structure. "continue" can only be used inside a "while", "for" or "do while" loop, while "break" can only be used inside those loops or a "switch" statement.

include files nested too deep ***(Assembler)***
Macro expansions and include file handling have filled up the assembler's internal stack. The maximum number of open macros and include files is 30.

included file * was converted to lower case ***(Preprocessor)***
The file specified to be included was not found, but a file with a lowercase version of the name of the file specified was found and used instead.

incompatible intermediate code version; should be * ***(Code Generator)***
The intermediate code file produced by P1 is not the correct version for use with this code generator. This is either that incompatible versions of one or more compilers have been installed in the same directory, or a temporary file error has occurred leading to corruption of a temporary file. Check the setting of the TEMP environment variable. If it refers to a long path name, change it to something shorter.

incomplete * record body: length = * ***(Linker)***
An object file contained a record with an illegal size. This probably means the file is truncated or not an object file.

incomplete ident record ***(Libr)***
The IDENT record in the object file was incomplete.

incomplete record ***(Objtohex, Libr)***
The object file passed to objtohex or the librarian is corrupted.

incomplete record: * ***(Linker)***
An object code record is incomplete. This is probably due to a corrupted or invalid object module. Re-compile the source file, watching for out of disk space errors etc.

incomplete record: type = * length = *
This message is produced by the DUMP or XSTRIP utilities and indicates that the object file is not a valid HI-TECH object file, or that it has been truncated, possibly due to running out of disk or RAMdisk space.

incomplete symbol record ***(Libr)***
The SYM record in the object file was incomplete.

inconsistent lineno tables ***(Cromwell)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

inconsistent storage class ***(Parser)***
A declaration has conflicting storage classes. Only one storage class should appear in a declaration.

inconsistent symbol tables ***(Cromwell)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

inconsistent type ***(Parser)***
Only one basic type may appear in a declaration, thus combinations like "int float" are illegal.

initialisation syntax ***(Parser)***
The initialisation of this object is syntactically incorrect. Check for the correct placement and number of braces and commas.

initializer in 'extern' declaration ***(Parser)***
A declaration containing the keyword "extern" has an initialiser. This overrides the "extern" storage class, since to initialise an object it is necessary to define (i.e. allocate storage for) it.

insufficient memory for macro def'n ***(Assembler)***
There is not sufficient memory to store a macro definition.

integer constant expected ***(Parser)***
A colon appearing after a member name in a structure declaration indicates that the member is a bitfield. An integral constant must appear after the colon to define the number of bits in the bitfield.

integer expression required ***(Parser)***
In an "enum" declaration, values may be assigned to the members, but the expression must evaluate to a constant of type "int".

integral argument required ***(Parser)***
An integral argument is required for this format specifier. Check the number and order of format specifiers and corresponding arguments.

integral type required ***(Parser)***
This operator requires operands that are of integral type only.

interrupt_level should be 0 to 7 ***(Parser)***
The pragma 'interrupt_level' must have an argument from 0 to 7.

invalid disable: * ***(Preprocessor)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

invalid format specifier or type modifier ***(Parser)***
The format specifier or modifier in the printf() style string is illegal for this particular format.

invalid hex file: *, line * ***(Cromwell)***
The specified Hex file contains an invalid line.

invalid number syntax ***(Assembler, Optimiser)***
The syntax of a number is invalid. This can be, e.g. use of 8 or 9 in an octal number, or other malformed numbers.

invalid operand size *(Assembler)*
The specified operand size (byte, word or double word) is invalid in this context.

invalid qualifier 'huge' on '*' - use huge only with pointers *(Code Generator)*
The huge type qualifier can be applied only to an object pointed to by a pointer.

invalid qualifier combination on * *(Code Generator)*
This qualifier combination is illegal, perhaps because it is contradictory.

invalid reloc value *(Assembler)*
The value for the RELOC psect flag must be in the range 0h to 10000h and, after that, any multiple of 10000h.

label identifier expected *(Parser)*
An identifier denoting a label must appear after "goto".

label not followed by : *(Optimiser)*
The optimizer has encountered a syntax error in its input.

lexical error *(Assembler, Optimiser)*
An unrecognized character or token has been seen in the input.

library * is badly ordered *(Linker)*
This library is badly ordered. It will still link correctly, but it will link faster if better ordered.

library file names should have .lib extension: * *(Libr)*
Use the .lib extension when specifying a library.

line does not have a newline on the end *(Parser)*
The last line in the file is missing the newline (linefeed, hex 0A) from the end. Some editors will create such files, which can cause problems for include files. The ANSI C standard requires all source files to consist of complete lines only.

line too long *(Optimiser)*
This line is too long. It will not fit into the compiler's internal buffers. It would require a line over 1000 characters long to do this, so it would normally only occur as a result of macro expansion.

local illegal outside macros *(Assembler)*
The "LOCAL" directive is only legal inside macros. It defines local labels that will be unique for each invocation of the macro.

local psect '*' conflicts with global psect of same name *(Linker)*
A local psect may not have the same name as a global psect.

logical type required ***(Parser)***
The expression used as an operand to "if", "while" statements or to boolean operators like ! and && must be a scalar integral type.

long argument required ***(Parser)***
A long argument is required for this format specifier. Check the number and order of format specifiers and corresponding arguments.

macro * wasn't defined ***(Preprocessor)***
A macro name specified in a -U option to the preprocessor was not initially defined, and thus cannot be undefined.

macro argument after * must be absolute ***(Assembler)***
The argument after * in a macro call must be absolute, as it must be evaluated at macro call time.

macro argument may not appear after local ***(Assembler)***
The list of labels after the directive "LOCAL" may not include any of the formal parameters to the macro.

macro expansions nested too deep ***(Assembler)***
Macro expansions in the assembler are nested too deep. The limit is 30 macros and include files nested at one time.

macro work area overflow ***(Preprocessor)***
The total length of a macro expansion has exceeded the size of an internal table. This table is normally 8192 bytes long. Thus any macro expansion must not expand into a total of more than 8K bytes.

member * redefined ***(Parser)***
This name of this member of the struct or union has already been used in this struct or union.

members cannot be functions ***(Parser)***
A member of a structure or a union may not be a function. It may be a pointer to a function. The correct syntax for a function pointer requires the use of parentheses to bind the star ("*") to the pointer name, e.g. "int (*name)();".

metaregister * can't be used directly ***(Code Generator)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

mismatched comparision ***(Code Generator)***
A comparison is being made between a variable or expression and a constant value which is not in the range of possible values for that expression, e.g. if you compare an unsigned character to the constant value 300, the result will always be false (not equal) since an unsigned character can NEVER equal 300. As an 8 bit value it can represent only 0-255.

misplaced '?' or ':', previous operator is * ***(Preprocessor)***
A colon operator has been encountered in a #if expression that does not match up with a corresponding ? operator. Check parentheses etc.

misplaced constant in #if ***(Preprocessor)***
A constant in a #if expression should only occur in syntactically correct places. This error is most probably caused by omission of an operator.

missing ')' ***(Parser)***
A closing parenthesis was missing from this expression.

missing '=' in class spec ***(Linker)***
A class spec needs an = sign, e.g. -Ctext=ROM

missing ']' ***(Parser)***
A closing square bracket was missing from this expression.

missing arg to -a ***(Parser)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

missing arg to -e ***(Linker)***
The error file name must be specified following the -e linker option.

missing arg to -i ***(Parser)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

missing arg to -j ***(Linker)***
The maximum number of errors before aborting must be specified following the -j linker option.

missing arg to -q ***(Linker)***
The -Q linker option requires the machine type for an argument.

missing arg to -u ***(Linker)***
The -U (undefine) option needs an argument, e.g. -U_symbol

missing arg to -w ***(Linker)***
The -W option (listing width) needs a numeric argument.

missing argument to 'pragma psect' ***(Parser)***
The pragma 'psect' requires an argument of the form oldname=newname where oldname is an existing psect name known to the compiler, and newname is the desired new name. Example: #pragma psect bss=battery

missing argument to 'pragma switch' ***(Parser)***
The pragma 'switch' requires an argument of auto, direct or simple.

missing basic type: int assumed ***(Parser)***
This declaration does not include a basic type, so int has been assumed. This declaration is not illegal, but it is preferable to include a basic type to make it clear what is intended.

missing key in avmap file ***(Linker)***
A file required for producing Avocet format symbol files is corrupted. Try re-installing the compiler.

missing memory key in avmap file ***(Linker)***
A file required for producing Avocet format symbol files is corrupted. Try re-installing the compiler.

missing name after pragma 'inline' ***(Parser)***
The 'inline' pragma has the syntax:

#pragma inline func_name

where func_name is the name of a function which is to be expanded to inline code. This pragma has no effect except on functions specially recognized by the code generator.

missing name after pragma 'printf_check' ***(Parser)***
The pragma 'printf_check', which enable printf style format string checking for a function, requires a function name, e.g.

#pragma printf_check sprintf

missing newline ***(Preprocessor)***
A new line is missing at the end of the line. Each line, including the last line, must have a new line at the end. This problem is normally introduced by editors.

missing number after % in -p option ***(Linker)***
The % operator in a -p option (for rounding boundaries) must have a number after it.

missing number after pragma 'pack' ***(Parser)***
The pragma 'pack' requires a decimal number as argument. For example

#pragma pack(1)

will prevent the compiler aligning structure members onto anything other than one byte boundaries. Use this with caution as some processors enforce alignment and will not operate correctly if word fetches are made on odd boundaries (e.g. 68000, 8096).

missing number after pragma interrupt_level ***(Parser)***
Pragma 'interrupt_level' requires an argument from 0 to 7.

missing processor name after -p ***(Cromwell)***
The -p option to cromwell must specify a processor.

mod by zero in #if, zero result assumed ***(Preprocessor)***
A modulus operation in a #if expression has a zero divisor. The result has been assumed to be zero.

module * defines no symbols ***(Libr)***
No symbols were found in the module's object file.

module has code below file base of * ***(Linker)***
This module has code below the address given, but the -C option has been used to specify that a binary output file is to be created that is mapped to this address. This would mean code from this module would have to be placed before the beginning of the file! Check for missing psect directives in assembler files.

multi-byte constant * isn't portable ***(Preprocessor)***
Multi-byte constants are not portable, and in fact will be rejected by later passes of the compiler.

multiple free: * ***(Code Generator)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

multiply defined symbol * ***(Assembler, Linker)***
This symbol has been defined in more than one place in this module.

nested #asm directive ***(Preprocessor)***
It is not legal to nest #asm directives. Check for a missing or misspelt #endasm directive.

nested comments ***(Preprocessor)***
This warning is issued when nested comments are found. A nested comment may indicate that a previous closing comment marker is missing or malformed.

no #asm before #endasm ***(Preprocessor)***
A #endasm operator has been encountered, but there was no previous matching #asm.

no arg to -o ***(Assembler)***
The assembler requires that an output file name argument be supplied after the "-O" option. No space should be left between the -O and the filename.

no case labels ***(Code Generator)***
There are no case labels in this switch statement.

no end record ***(Linker)***
This object file has no end record. This probably means it is not an object file.

no end record found ***(Linker)***
An object file did not contain an end record. This probably means the file is corrupted or not an object file.

no file arguments *(Assembler)*
The assembler has been invoked without any file arguments. It cannot assemble anything.

no identifier in declaration *(Parser)*
The identifier is missing in this declaration. This error can also occur where the compiler has been confused by such things as missing closing braces.

no input files specified *(Cromwell)*
Cromwell must have an input file to convert.

no memory for string buffer *(Parser)*
P1 was unable to allocate memory for the longest string encountered, as it attempts to sort and merge strings. Try reducing the number or length of strings in this module.

no output file format specified *(Cromwell)*
The output format must be specified to Cromwell.

no psect specified for function variable/argument allocation *(Linker)*
This is probably caused by omission of correct run-time startoff module. Check the linker arguments, or "Object Files..." in HPD.

no room for arguments *(Preprocessor, Parser, Code Generator, Linker, Objtohex)*
The code generator could not allocate any more memory. Try increasing the size of available memory.

no space for macro def'n *(Assembler)*
The assembler has run out of memory.

no start record: entry point defaults to zero *(Linker)*
None of the object files passed to the linker contained a start record. The start address of the program has been set to zero. This may be harmless, but it is recommended that you define a start address in your startup module by using the "END" directive.

no. of arguments redeclared *(Parser)*
The number of arguments in this function declaration does not agree with a previous declaration of the same function.

nodecount = * *(Code Generator)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

non-constant case label *(Code Generator)*
A case label in this switch statement has a value which is not a constant.

non-prototyped function declaration: * *(Parser)*
A function has been declared using old-style (K&R) arguments. It is preferable to use prototype declarations for all functions. If the function has no arguments, declare it as e.g. "int func(void)".

non-scalar types can't be converted ***(Parser)***
You can't convert a structure, union or array to anything else. You can convert a pointer to one of those things, so perhaps you left out an ampersand ("&").

non-void function returns no value ***(Parser)***
A function that is declared as returning a value has a "return" statement that does not specify a return value.

not a member of the struct/union * ***(Parser)***
This identifier is not a member of the structure or union type with which it used here.

not a variable identifier: * ***(Parser)***
This identifier is not a variable; it may be some other kind of object, e.g. a label.

not an argument: * ***(Parser)***
This identifier that has appeared in a K&R stype argument declarator is not listed inside the parentheses after the function name. Check spelling.

null format name ***(Cromwell)***
The -I or -O option to Cromwell must specify a file format.

nxtuse() : unknown op * ***(Optimiser)***
Internal error - Contact HI-TECH Software.

object code version is greater than * ***(Linker)***
The object code version of an object module is higher than the highest version the linker is known to work with. Check that you are using the correct linker.

object file is not absolute ***(Objtohex)***
The object file passed to objtohex has relocation items in it. This may indicate it is the wrong object file, or that the linker or objtohex have been given invalid options.

only functions may be qualified interrupt ***(Parser)***
The qualifier "interrupt" may not be applied to anything except a function.

only functions may be void ***(Parser)***
A variable may not be "void". Only a function can be "void".

only lvalues may be assigned to or modified ***(Parser)***
Only an lvalue (i.e. an identifier or expression directly denoting addressable storage) can be assigned to or otherwise modified. A typecast does not yield an lvalue. To store a value of different type into a variable, take the address of the variable, convert it to a pointer to the desired type, then dereference that pointer, e.g. "*(int *)&x = 1" is legal whereas "(int)x = 1" is not.

only modifier l valid with this format *(Parser)*
The only modifier that is legal with this format is l (for long).

only modifiers h and l valid with this format *(Parser)*
Only modifiers h (short) and l (long) are legal with this printf() format specifier.

only register storage class allowed *(Parser)*
The only storage class allowed for a function parameter is "register".

oops! -ve number of nops required! *(Assembler)*
An internal error has occurred. Contact HI-TECH.

operand error *(Assembler, Optimiser)*
The operand to this opcode is invalid. Check you assembler reference manual for the proper form of operands for this instruction.

operands of * not same pointer type *(Parser)*
The operands of this operator are of different pointer types. This probably means you have used the wrong pointer, but if the code is actually what you intended, use a typecast to suppress the error message.

operands of * not same type *(Parser)*
The operands of this operator are of different pointer. This probably means you have used the wrong variable, but if the code is actually what you intended, use a typecast to suppress the error message.

operator * in incorrect context *(Preprocessor)*
An operator has been encountered in a #if expression that is incorrectly placed, e.g. two binary operators are not separated by a value.

out of far memory *(Code Generator)*
The compiler has run out of far memory. Try removing TSR's etc. If your system supports EMS memory, the compiler will be able to use up to 64K of this, so if it is not enable, try enabling EMS.

out of memory *(Code Generator, Assembler, Optimiser)*
The compiler has run out of memory. If you have unnecessary TSRs loaded, remove them. If you are running the compiler from inside another program, try running it directly from the command prompt. Similarly, if you are using HPD, try using the command line compiler driver instead.

out of memory allocating * blocks of * *(Linker)*
Memory was required to extend an array but was unavailable.

out of memory for strings *(Optimiser)*
The optimizer has run out of memory to store strings.

out of near memory ***(Code Generator)***
The compiler has run out of near memory. This is probably due to too many symbol names. Try splitting the program up, or reducing the number of unused symbols in header files etc.

out of space in macro * arg expansion ***(Preprocessor)***
A macro argument has exceeded the length of an internal buffer. This buffer is normally 4096 bytes long.

out-of-range case label * ***(Code Generator)***
This case label is not a value that the controlling expression can yield, and thus this label will never be selected.

output file cannot be also an input file ***(Linker)***
The linker has detected an attempt to write its output file over one of its input files. This cannot be done, because it needs to simultaneously read and write input and output files.

page width must be >= * ***(Assembler)***
The listing page width must be at least * characters. Any less will not allow a properly formatted listing to be produced.

phase error ***(Assembler)***
The assembler has calculated a different value for a symbol on two different passes. This is probably due to bizarre use of macros or conditional assembly.

phase error in macro args ***(Assembler)***
The assembler has detected a difference in the definition of a symbol on the first and a subsequent pass.

pointer required ***(Parser)***
A pointer is required here. This often means you have used "->" with a structure rather than a structure pointer.

pointer to * argument required ***(Parser)***
A pointer argument is required for this format specifier. Check the number and order of format specifiers and corresponding arguments.

pointer to non-static object returned ***(Parser)***
This function returns a pointer to a non-static (e.g. automatic) variable. This is likely to be an error, since the storage associated with automatic variables becomes invalid when the function returns.

portion of expression has no effect ***(Code Generator)***
Part of this expression has no side effects, and no effect on the value of the expression.

possible pointer truncation ***(Parser)***
A pointer qualified "far" has been assigned to a default pointer or a pointer qualified "near", or a default pointer has been assigned to a pointer qualified "near". This may result in truncation of the pointer and loss of information, depending on the memory model in use.

preprocessor assertion failure ***(Preprocessor)***
The argument to a preprocessor #assert directive has evaluated to zero. This is a programmer induced error.

probable missing '}' in previous block ***(Parser)***
The compiler has encountered what looks like a function or other declaration, but the preceding function has not been ended with a closing brace. This probably means that a closing brace has been omitted from somewhere in the previous function, although it may well not be the last one.

psect * cannot be in classes * ***(Linker)***
A psect cannot be in more than one class. This is either due to assembler modules with conflicting class= options, or use of the -C option to the linker.

psect * memory delta redefined: */* ***(Linker)***
A global psect has been defined with two different deltas.

psect * memory space redefined: */* ***(Linker)***
A global psect has been defined in two different memory spaces. Either rename one of the psects or, if they are the same psect, place them in the same memory space using the SPACE psect flag.

psect * not loaded on * boundary ***(Linker)***
This psect has a relocatability requirement that is not met by the load address given in a -P option. For example if a psect must be on a 4K byte boundary, you could not start it at 100H.

psect * not relocated on * boundary ***(Linker)***
This psect is not relocated on the required boundary. Check the relocatability of the psect and correct the -p option. if necessary.

psect * not specified in -p option ***(Linker)***
This psect was not specified in a -P or -A option to the linker. It has been linked at the end of the program, which is probably not where you wanted it.

psect * re-orged ***(Linker)***
This psect has had its start address specified more than once.

psect * selector value redefined ***(Linker)***
The selector value for this psect has been defined more than once.

psect * type redefined: * ***(Linker)***
This psect has had its type defined differently by different modules. This probably means you are trying to link incompatible object modules, e.g. linking 386 flat model code with 8086 real mode code.

psect alignment redefined ***(Assembler)***
The psect alignment has already been defined using the psect ALIGN flag elsewhere.

psect exceeds address limit: * ***(Linker)***
The maximum address of the psect exceeds the limit placed on it using the LIMIT psect flag.

psect exceeds max size: * ***(Linker)***
The psect has more bytes in it than the maximum allowed as specified using the SIZE psect flag.

psect is absolute: * ***(Linker)***
This psect is absolute and should not have an address specified in a -P option.

psect limit redefined ***(Assembler)***
The psect limit has already been defined using the psect LIMIT flag elsewhere.

psect may not be local and global ***(Assembler)***
A psect may not be declared to be local if it has already been declared to be (default) global.

psect origin multiply defined: * ***(Linker)***
The origin of this psect is defined more than once.

psect property redefined ***(Assembler)***
A property of a psect has been defined in more than place to be different.

psect reloc redefined ***(Assembler)***
The relocatability of this psect has been defined differently in two or more places.

psect selector redefined ***(Linker)***
The selector associated with this psect has been defined differently in two or more places.

psect size redefined ***(Assembler)***
The maximum size of this psect has been defined differently in two or more places.

psect space redefined ***(Assembler)***
The psect space has already been defined using the psect SPACE flag elsewhere.

qualifiers redeclared ***(Parser)***
This function has different qualifiers in different declarations.

range check too complex ***(Assembler)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

read error on * ***(Linker)***
The linker encountered an error trying to read this file.

record too long ***(Objtohex)***
This indicates that the object file is not a valid HI-TECH object file.

record too long: * ***(Linker)***
An object file contained a record with an illegal size. This probably means the file is corrupted or not an object file.

recursive function calls: ***(Linker)***
These functions (or function) call each other recursively. One or more of these functions has statically allocated local variables (compiled stack). Either use the reentrant keyword (if supported with this compiler) or recode to avoid recursion.

recursive macro definition of * ***(Preprocessor)***
The named macro has been defined in such a manner that expanding it causes a recursive expansion of itself!

redefining macro * ***(Preprocessor)***
The macro specified is being redefined, to something different to the original definition. If you want to deliberately redefine a macro, use #undef first to remove the original definition.

redundant & applied to array ***(Parser)***
The address operator "&" has been applied to an array. Since using the name of an array gives its address anyway, this is unnecessary and has been ignored.

refc == 0 ***(Assembler, Optimiser)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

regused - bad arg to g ***(Code Generator)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

relocation error ***(Assembler, Optimiser)***
It is not possible to add together two relocatable quantities. A constant may be added to a relocatable value, and two relocatable addresses in the same psect may be subtracted. An absolute value must be used in various places where the assembler must know a value at assembly time.

relocation offset * out of range * ***(Linker)***
An object file contained a relocation record with a relocation offset outside the range of the preceding text record. This means the object file is probably corrupted.

relocation too complex ***(Assembler)***
The complex relocation in this expression is too big to be inserted into the object file.

remsym error ***(Assembler)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

replace what ? ***(Libr)***
The librarian requires one or more modules to be listed for replacement when using the ’r’ key.

rept argument must be >= 0 ***(Assembler)***
The argument to a "REPT" directive must be greater than zero.

restore without matching save ***(Assembler)***
The RESTORE assembler control directive has been used without a preceding SAVE assembler control directive.

save/restore too deep ***(Assembler)***
Too many SAVE assembler control directives have been used.

seek error: * ***(Linker)***
The linker could not seek when writing an output file.

segment * overlaps segment * ***(Linker)***
The named segments have overlapping code or data. Check the addresses being assigned by the "-P" option.

signatures do not match: * ***(Linker)***
The specified function has different signatures in different modules. This means it has been declared differently, e.g. it may have been prototyped in one module and not another. Check what declarations for the function are visible in the two modules specified and make sure they are compatible.

signed bitfields not supported ***(Parser)***
Only unsigned bitfields are supported. If a bitfield is declared to be type "int", the compiler still treats it as unsigned.

simple integer expression required ***(Parser)***
A simple integral expression is required after the operator "@", used to associate an absolute address with a variable.

simple type required for * ***(Parser)***
A simple type (i.e. not an array or structure)is required as an operand to this operator.

sizeof external array * is zero ***(Parser)***
The sizeof an external array evaluates to zero. This is probably due to the array not having an explicit dimension in the extern declaration.

sizeof yields 0 ***(Code Generator)***
The code generator has taken the size of an object and found it to be zero. This almost certainly indicates an error in your declaration of a pointer, e.g. you may have declared a pointer to a zero length array. In general, pointers to arrays are of little use. If you require a pointer to an array of objects of unknown length, you only need a pointer to a single object that can then be indexed or incremented.

sizer required after dot *(Assembler)*
The size of the operand is required. For example, MOV.w indicates that data of word size is to be moved. 'w' is the 'sizer'.

ssel still set * at function end ? *(Code Generator)*
Don't know

static object has zero size: * *(Code Generator)*
A static object has been declared, but has a size of zero.

storage class illegal *(Parser)*
A structure or union member may not be given a storage class. Its storage class is determined by the storage class of the structure.

storage class redeclared *(Parser)*
A variable or function has been re-declared with a different storage class. This can occur where there are two conflicting declarations, or where an implicit declaration is followed by an actual declaration.

strange character * after ## *(Preprocessor)*
A character has been seen after the token catenation operator ## that is neither a letter nor a digit. Since the result of this operator must be a legal token, the operands must be tokens containing only letters and digits.

strange character after # * *(Preprocessor)*
There is an unexpected character after #.

string concatenation across lines *(Parser)*
Strings on two lines will be concatenated. Check that this is the desired result.

string expected *(Parser)*
The operand to an "asm" statement must be a string enclosed in parentheses.

string lookup failed in coff:get_string() *(Cromwell)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

struct/union member expected *(Parser)*
A structure or union member name must follow a dot (".") or arrow ("->").

struct/union redefined: * *(Parser)*
A structure or union has been defined more than once.

struct/union required *(Parser)*
A structure or union identifier is required before a dot (".").

struct/union tag or '{' expected ***(Parser)***
An identifier denoting a structure or union or an opening brace must follow a "struct" or "union" keyword.

symbol * cannot be global ***(Linker)***
There is an error in an object file, where a local symbol has been declared global. This is either an invalid object file, or an internal error in the linker. Try recreating the object file.

symbol * has erroneous psect: * ***(Linker)***
There is an error in an object file, where a symbol has an invalid psect. This is either an invalid object file, or an internal error in the linker. Try recreating the object file.

symbol * is not external ***(Assembler)***
A symbol has been declared as EXTRN but is also defined in the current module.

symbol * not defined in #undef ***(Preprocessor)***
The symbol supplied as argument to #undef was not already defined. This is a warning only, but could be avoided by including the #undef in a #ifdef ... #endif block.

syntax error ***(Assembler, Optimiser)***
A syntax error has been detected. This could be caused a number of things.

syntax error in -a spec ***(Linker)***
The -A spec is invalid. A valid -A spec should be something like:

-AROM=1000h-1FFFh

syntax error in checksum list ***(Linker)***
There is a syntax error in a checksum list read by the linker. The checksum list is read from standard input by the linker, in response to an option. Re-read the manual on checksum list.

text does not start at 0 ***(Linker)***
Code in some things must start at zero. Here it doesn't.

text offset too low ***(Linker)***
You aren't likely to see this error. Rhubarb!

text record has bad length: * ***(Linker)***
There is an error in an object file. This is either an invalid object file, or an internal error in the linker. Try recreating the object file.

text record has length too small: * ***(Linker)***
This indicates that the object file is not a valid HI-TECH object file.

the reserved 0x7F (delete) character has been used ***(Assembler)***
A delete character has been detected in the assembly source file. The delete character is reserved by the assembler and must be removed.

this function too large - try reducing level of optimization ***(Code Generator)***
A large function has been encountered when using a -Og (global optimization) switch. Try re-compiling without the global optimization, or reduce the size of the function.

this is a struct ***(Parser)***
This identifier following a "union" or "enum" keyword is already the tag for a structure, and thus should only follow the keyword "struct".

this is a union ***(Parser)***
This identifier following a "struct" or "enum" keyword is already the tag for a union, and thus should only follow the keyword "union".

this is an enum ***(Parser)***
This identifier following a "struct" or "union" keyword is already the tag for an enumerated type, and thus should only follow the keyword "enum".

too few arguments ***(Parser)***
This function requires more arguments than are provided in this call.

too few arguments for format string ***(Parser)***
There are too few arguments for this format string. This would result in a garbage value being printed or converted at run time.

too many (*) enumeration constants ***(Parser)***
There are too many enumeration constants in an enumerated type. The maximum number of enumerated constants allowed in an enumerated type is 512.

too many (*) structure members ***(Parser)***
There are too many members in a structure or union. The maximum number of members allowed in one structure or union is 512.

too many address spaces - space * ignored ***(Linker)***
The limit to the number of address spaces is currently 16.

too many arguments ***(Parser)***
This function does not accept as many arguments as there are here.

too many arguments for format string ***(Parser)***
There are too many arguments for this format string. This is harmless, but may represent an incorrect format string.

too many arguments for macro ***(Preprocessor)***
A macro may only have up to 31 parameters, as per the C Standard.

too many arguments in macro expansion ***(Preprocessor)***
There were too many arguments supplied in a macro invocation. The maximum number allowed is 31.

too many cases in switch ***(Code Generator)***
There are too many case labels in this switch statement. The maximum allowable number of case labels in any one switch statement is 511.

too many comment lines - discarding ***(Assembler)***
The compiler is generating assembler code with embedded comments, but this function is so large that an excessive number of source line comments are being generated. This has been suppressed so that the optimizer will not run out of memory loading comment lines.

too many errors ***(Preprocessor, Parser, Code Generator, Assembler, Linker)***
There were so many errors that the compiler has given up. Correct the first few errors and many of the later ones will probably go away.

too many file arguments. usage: cpp [input [output]] ***(Preprocessor)***
CPP should be invoked with at most two file arguments.

too many files in coff file ***(Cromwell)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

too many include directories ***(Preprocessor)***
A maximum of 7 directories may be specified for the preprocessor to search for include files.

too many initializers ***(Parser)***
There are too many initializers for this object. Check the number of initializers against the object definition (array or structure).

too many input files ***(Cromwell)***
To many input files have been specified to be converted by Cromwell.

too many macro parameters ***(Assembler)***
There are too many macro parameters on this macro definition.

too many nested #* statements ***(Preprocessor)***
#if, #ifdef etc. blocks may only be nested to a maximum of 32.

too many nested #if statements ***(Preprocessor)***
#if, #ifdef etc. blocks may only be nested to a maximum of 32.

too many operands *(Optimiser)*
There are too many operands to this instruction.

too many output files *(Cromwell)*
To many output file formats have been specified to Cromwell.

too many psect class specifications *(Linker)*
There are too many psect class specifications (-C options)

too many psect pragmas *(Code Generator)*
Too many "pragma psect" directives have been used.

too many psects *(Assembler)*
There are too many psects! Boy, what a program!

too many qualifier names *(Parser)*
There are too many qualifier names specified.

too many relocation items *(Objtohex)*
Objtohex filled up a table. This program is just way too complex!

too many segment fixups *(Objtohex)*
There are too many segment fixups in the object file given to objtohex.

too many segments *(Objtohex)*
There are too many segments in the object file given to objtohex.

too many symbols *(Assembler)*
There are too many symbols for the assemblers symbol table. Reduce the number of symbols in your program. If it is the linker producing this error, suggest changing some global to local symbols.

too many symbols (*) *(Linker)*
There are too many symbols in the symbol table, which has a limit of * symbols. Change some global symbols to local symbols to reduce the number of symbols.

too many symbols in * *(Optimiser)*
There are too many symbols in the specified function. Reduce the size of the function.

too many temporary labels *(Assembler)*
There are too many temporary labels in this assembler file. The assembler allows a maximum of 2000 temporary labels.

too much indirection *(Parser)*
A pointer declaration may only have 16 levels of indirection.

too much pushback *(Preprocessor)*
This error should not occur, and represents an internal error in the preprocessor.

trap number must be a constant *(Code Generator)*
The argument to a software TRAP must be a constant in the range 0 to 15.

type conflict *(Parser)*
The operands of this operator are of incompatible types.

type modifier already specified *(Parser)*
This type modifier has already be specified in this type.

type modifiers not valid with this format *(Parser)*
Type modifiers may not be used with this format.

type redeclared *(Parser)*
The type of this function or object has been redeclared. This can occur because of two incompatible declarations, or because an implicit declaration is followed by an incompatible declaration.

type specifier reqd. for proto arg *(Parser)*
A type specifier is required for a prototyped argument. It is not acceptable to just have an identifier.

unable to open list file * *(Linker)*
The named list file could not be opened.

unbalanced paren's, op is * *(Preprocessor)*
The evaluation of a #if expression found mismatched parentheses. Check the expression for correct parenthesisation.

undefined *: * *(Parser)*
This is an internal compiler error. Contact HI-TECH Software technical support with details.

undefined enum tag: * *(Parser)*
This enum tag has not been defined.

undefined identifier: * *(Parser)*
This symbol has been used in the program, but has not been defined or declared. Check for spelling errors.

undefined label: *
This label has been used but not defined at this point.

undefined public symbol * *(Assembler)*
A symbol has been declared PUBLIC but has not been defined.anywhere in the module.

undefined shift (* bits) ***(Code Generator)***
An attempt has been made to shift a value by a number of bits equal to or greater than the number of bits in the data type, e.g. shifting a long by 32 bits. This will produce an undefined result on many processors. This is non-portable code and is flagged as having undefined results by the C Standard.

undefined struct/union ***(Parser)***
This structure or union tag is undefined. Check spelling etc.

undefined struct/union: * ***(Parser)***
The specified structure or union tag is undefined. Check spelling etc.

undefined symbol * ***(Assembler)***
The named symbol is not defined, and has not been specified "GLOBAL".

undefined symbol * in #if, 0 used ***(Preprocessor)***
A symbol on a #if expression was not a defined preprocessor macro. For the purposes of this expression, its value has been taken as zero.

undefined symbol in fnaddr record: * ***(Linker)***
The linker has found an undefined symbol in the fnaddr record for a non-reentrant function.

undefined symbol in fnbreak record: * ***(Linker)***
The linker has found an undefined symbol in the fnbreak record for a non-reentrant function.

undefined symbol in fncall record: * ***(Linker)***
The linker has found an undefined symbol in the fncall record for a non-reentrant function.

undefined symbol in fnindir record: * ***(Linker)***
The linker has found an undefined symbol in the fnindir record for a non-reentrant function.

undefined symbol in fnroot record: * ***(Linker)***
The linker has found an undefined symbol in the fnroot record for a non-reentrant function.

undefined symbol in fnsize record: * ***(Linker)***
The linker has found an undefined symbol in the fnsize record for a non-reentrant function.

undefined symbol: ***(Assembler, Linker)***
The symbol following is undefined at link time. This could be due to spelling error, or failure to link an appropriate module.

undefined symbols: ***(Linker)***
A list of symbols follows that were undefined at link time.

undefined temporary label *(Assembler)*
A temporary label has been referenced that is not defined. Note that a temporary label must have a number >= 0.

undefined variable: * *(Parser)*
This variable has been used but not defined at this point.

unexpected end of file *(Linker)*
This probably means an object file has been truncated because of a lack of disk space.

unexpected eof *(Parser)*
An end-of-file was encountered unexpectedly. Check syntax.

unexpected text in #control line ignored *(Preprocessor)*
This warning occurs when extra characters appear on the end of a control line, e.g.

#endif something

The "something" will be ignored, but a warning is issued. It is preferable (and in accordance with Standard C) to enclose the "something" as a comment, e.g.

#endif /* something */

unexpected \ in #if *(Preprocessor)*
The backslash is incorrect in the #if statement.

unknown 'with' psect referenced by psect * *(Linker)*
The specified psect has been placed with a psect using the psect 'with' flag. The psect it has been placed with does not exist.

unknown addressing mode * *(Assembler, Optimiser)*
An unknown addressing mode was used in the assembly file.

unknown argument to 'pragma switch': * *(Code Generator)*
The '#pragma switch' directive has been used with an invalid switch code generation method. Possible arguments are: auto, simple and direct.

unknown complex operator * *(Linker)*
There is an error in an object file. This is either an invalid object file, or an internal error in the linker. Try recreating the object file.

unknown directive *(Assembler)*
An unknown assembler control directive was used.

unknown fnrec type * *(Linker)*
This indicates that the object file is not a valid HI-TECH object file.

unknown format name '*' ***(Cromwell)***
The output format specified to Cromwell is unknown.

unknown option * ***(Preprocessor)***
This option to the preprocessor is not recognized.

unknown pragma * ***(Parser)***
An unknown pragma directive was encountered.

unknown predicate * ***(Code Generator)***
Internal error - Contact HI-TECH.

unknown psect ***(Optimiser)***
The assembler file read by the optimizer has an unknown psect.

unknown psect: * ***(Linker, Optimiser)***
This psect has been listed in a -P option, but is not defined in any module within the program.

unknown qualifier '*' given to -a ***(Parser)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

unknown qualifier '*' given to -i ***(Parser)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

unknown record type: * ***(Linker)***
An invalid object module has been read by the linker. It is either corrupted or not an object file.

unknown register name * ***(Linker)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

unknown segment ***(Optimiser)***
This is an internal optimiser error. Contact HI-TECH Software technical support.

unknown symbol type * ***(Linker)***
The symbol type encountered is unknown to this linker. Check that the correct linker is being used.

unreachable code ***(Parser)***
This section of code will never be executed, because there is no execution path by which it could be reached. Look for missing "break" statements inside a control structure like "while" or "for".

unreasonable matching depth ***(Code Generator)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

unrecognized option to -z: * ***(Code Generator)***
The code generator has been passed a -Z option it does not understand. This should not happen if it is invoked with the standard driver.

unrecognized qualifer name after 'strings' ***(Parser)***
The pragma 'strings' requires a list of valid qualifier names. For example

#pragma strings const code

would add const and code to the current string qualifiers. If no qualifiers are specified, all qualification will be removed from subsequent strings. The qualifier names must be recognized by the compiler.

unterminated #if[n][def] block from line * ***(Preprocessor)***
A #if or similar block was not terminated with a matching #endif. The line number is the line on which the #if block began.

unterminated comment in included file ***(Preprocessor)***
Comments begun inside an included file must end inside the included file.

unterminated macro arg ***(Assembler)***
An argument to a macro is not terminated. Note that angle brackets ("< >") are used to quote macro arguments.

unterminated string ***(Assembler, Optimiser)***
A string constant appears not to have a closing quote missing.

unterminated string in macro body ***(Preprocessor, Assembler)***
A macro definition contains a string that lacks a closing quote.

unused constant: * ***(Parser)***
This enumerated constant is never used. Maybe it isn't needed at all.

unused enum: * ***(Parser)***
This enumerated type is never used. Maybe it isn't needed at all.

unused label: * ***(Parser)***
This label is never used. Maybe it isn't needed at all.

unused member: * ***(Parser)***
This structure member is never used. Maybe it isn't needed at all.

unused structure: * ***(Parser)***
This structure tag is never used. Maybe it isn't needed at all.

unused typedef: * ***(Parser)***
This typedef is never used. Maybe it isn't needed at all.

unused union: * ***(Parser)***
This union type is never used. Maybe it isn't needed at all.

unused variable declaration: * ***(Parser)***
This variable is never used. Maybe it isn't needed at all.

unused variable definition: * ***(Parser)***
This variable is never used. Maybe it isn't needed at all.

upper case #include files are non-portable ***(Preprocessor)***
When using DOS, the case of an #include file does not matter. In other operating systems the case is significant.

variable may be used before set: * ***(Code Generator)***
This variable may be used before it has been assigned a value. Since it is an auto variable, this will result in it having a random value.

void function cannot return value ***(Parser)***
A void function cannot return a value. Any "return" statement should not be followed by an expression.

while expected ***(Parser)***
The keyword "while" is expected at the end of a "do" statement.

work buffer overflow doing * ## ***(Preprocessor)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

work buffer overflow: * ***(Preprocessor)***
This is an internal compiler error. Contact HI-TECH Software technical support with details.

write error (out of disk space?) * ***(Linker)***
Probably means that the hard disk is full.

write error on * ***(Assembler, Linker, Cromwell)***
A write error occurred on the named file. This probably means you have run out of disk space.

write error on object file ***(Assembler)***
An error was reported when the assembler was attempting to write an object file. This probably means there is not enough disk space.

wrong number of macro arguments for * - * instead of * ***(Preprocessor)***
A macro has been invoked with the wrong number of arguments.

Library functions

The functions within the HI-TECH C XA compiler library are listed in this chapter. Each entry begins with the name of the function. This is followed by information analysed into the following headings.

Synopsis

This is the C definition of the function, and the header file in which it is declared.

Description

This is a narrative description of the function and its purpose.

Example

This is an example of the use of the function. It is usually a complete small program that illustrates the function.

Data types

If any special data types (structures etc.) are defined for use with the function, they are listed here with their C definition. These data types will be defined in the header file given under heading - Synopsis.

See also

This refers you to any allied functions.

Return value

The type and nature of the return value of the function, if any, is given. Information on error returns is also included

Only those headings which are relevant to each function are used.

ABS

Synopsis

```
#include <stdlib.h>

int abs (int j)
```

Description

The **abs()** function returns the absolute value of **j**.

Example

```
#include <stdio.h>
#include <stdlib.h>

void
main (void)
{
    int a = -5;

    printf("The absolute value of %d is %d\n", a, abs(a));
}
```

Return Value

The absolute value of **j**.

ACOS

Synopsis

```
#include <math.h>

double acos (double f)
```

Description

The **acos()** function implements the converse of cos(), i.e. it is passed a value in the range -1 to +1, and returns an angle in radians whose cosine is equal to that value.

Example

```
#include <math.h>
#include <stdio.h>

    /* Print acos() values for -1 to 1 in degrees. */

void
main (void)
{
    float i, a;

    for(i = -1.0; i < 1.0 ; i += 0.1) {
        a = acos(i)*180.0/3.141592;
        printf("acos(%f) = %f degrees\n", i, a);
    }
}
```

See Also

sin(), cos(), tan(), asin(), atan(), atan2()

Return Value

An angle in radians, in the range 0 to π. Where the argument value is outside the domain -1 to 1, the return value will be zero.

ASCTIME

Synopsis

```
#include <time.h>

char * asctime (struct tm * t)
```

Description

The **asctime()** function takes the time broken down into the **struct tm** structure, pointed to by its argument, and returns a 26 character string describing the current date and time in the format:

Sun Sep 16 01:03:52 1973\n\0

Note the *newline* at the end of the string. The width of each field in the string is fixed. The example gets the current time, converts it to a **struct tm** pointer with `localtime()`, it then converts this to ASCII and prints it. The `time()` function will need to be provided by the user (see `time()` for details).

Example

```
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t clock;
    struct tm * tp;

    time(&clock);
    tp = localtime(&clock);
    printf("%s", asctime(tp));
}
```

See Also

`ctime(), gmtime(), localtime(), time()`

Return Value

A pointer to the string.

Note

The example will require the user to provide the `time()` routine as it cannot be supplied with the compiler. See `time()` for more details.

Data Types

```
struct tm {
     int tm_sec;
     int tm_min;
     int tm_hour;
     int tm_mday;
     int tm_mon;
     int tm_year;
     int tm_wday;
     int tm_yday;
     int tm_isdst;
};
```

ASIN

Synopsis

```
#include <math.h>

double asin (double f)
```

Description

The **`asin()`** function implements the converse of `sin()`, i.e. it is passed a value in the range -1 to +1, and returns an angle in radians whose sine is equal to that value.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    float i, a;

    for(i = -1.0; i < 1.0 ; i += 0.1) {
        a = asin(i)*180.0/3.141592;
        printf("asin(%f) = %f degrees\n", i, a);
    }
}
```

See Also

`sin(), cos(), tan(), acos(), atan(), atan2()`

Return Value

An angle in radians, in the range $-\pi/2$ to $+\pi/2$. Where the argument value is outside the domain -1 to 1, the return value will be zero.

ASSERT

Synopsis

```
#include <assert.h>

void assert (int e)
```

Description

This macro is used for debugging purposes; the basic method of usage is to place assertions liberally throughout your code at points where correct operation of the code depends upon certain conditions being true initially. An **`assert()`** routine may be used to ensure at run time that an assumption holds true. For example, the following statement asserts that the pointer `tp` is not equal to `NULL`:

```
assert(tp);
```

If at run time the expression evaluates to false, the program will abort with a message identifying the source file and line number of the assertion, and the expression used as an argument to it. A fuller discussion of the uses of **`assert()`** is impossible in limited space, but it is closely linked to methods of proving program correctness.

Example

```
void
ptrfunc (struct xyz * tp)
{
    assert(tp != 0);
}
```

Note

When required for ROM based systems, the underlying routine `_fassert(...)` will need to be implemented by the user.

ATAN

Synopsis

```
#include <math.h>

double atan (double x)
```

Description

This function returns the arc tangent of its argument, i.e. it returns an angle `e` in the range -π/2 to π/2 such that `tan(e) == x`.

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    printf("%f\n", atan(1.5));
}
```

See Also

`sin(), cos(), tan(), asin(), acos(), atan2()`

Return Value

The arc tangent of its argument.

ATAN2

Synopsis

```
#include <math.h>

double atan2 (double y, double x)
```

Description

This function returns the arc tangent of **y/x**, using the sign of both arguments to determine the quadrant of the return value.

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    printf("%f\n", atan2(1.5, 1));
}
```

See Also

sin(), cos(), tan(), asin(), acos(), atan()

Return Value

The arc tangent of **y/x** in the range -π to +π radians. If both **y** and **x** are zero, a domain error occurs and zero is returned.

ATOF

Synopsis

```
#include <stdlib.h>

double atof (const char * s)
```

Description

The **atof()** function scans the character string passed to it, skipping leading blanks. It then converts an ASCII representation of a number to a double. The number may be in decimal, normal floating point or scientific notation.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[80];
    double i;

    gets(buf);
    i = atof(buf);
    printf("Read %s: converted to %f\n", buf, i);
}
```

See Also

atoi(), atol()

Return Value

A double precision floating point number. If no number is found in the string, 0.0 will be returned.

ATOI

Synopsis

```
#include <stdlib.h>

int atoi (const char * s)
```

Description

The **atoi()** function scans the character string passed to it, skipping leading blanks and reading an optional sign. It then converts an ASCII representation of a decimal number to an integer.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[80];
    int i;

    gets(buf);
    i = atoi(buf);
    printf("Read %s: converted to %d\n", buf, i);
}
```

See Also

xtoi(), atof(), atol()

Return Value

A signed integer. If no number is found in the string, 0 will be returned.

ATOL

Synopsis

```
#include <stdlib.h>

long atol (const char * s)
```

Description

The **atol()** function scans the character string passed to it, skipping leading blanks. It then converts an ASCII representation of a decimal number to a long integer.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[80];
    long i;

    gets(buf);
    i = atol(buf);
    printf("Read %s: converted to %ld\n", buf, i);
}
```

See Also

atoi(), atof()

Return Value

A long integer. If no number is found in the string, 0 will be returned.

BSEARCH

Synopsis

```
#include <stdlib.h>

void * bsearch (const void * key, void * base, size_t n_memb,
          size_t size, int (*compar)(const void *, const void *))
```

Description

The **bsearch()** function searches a sorted array for an element matching a particular key. It uses a binary search algorithm, calling the function pointed to by **compar** to compare elements in the array.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

struct value {
    char name[40];
    int value;
} values[100];

int
val_cmp (const void * p1, const void * p2)
{
    return strcmp(((const struct value *)p1)->name,
            ((const struct value *)p2)->name);
}

void
main (void)
{
    char inbuf[80];
    int i;
    struct value * vp;

    i = 0;
    while(gets(inbuf)) {
        sscanf(inbuf,"%s %d", values[i].name, &values[i].value);
        i++;
    }
    qsort(values, i, sizeof values[0], val_cmp);
```

```
    vp = bsearch("fred", values, i, sizeof values[0], val_cmp);
    if(!vp)
        printf("Item 'fred' was not found\n");
    else
        printf("Item 'fred' has value %d\n", vp->value);
}
```

See Also

`qsort()`

Return Value

A pointer to the matched array element (if there is more than one matching element, any of these may be returned). If no match is found, a null pointer is returned.

Note

The comparison function must have the correct prototype.

CALLOC

Synopsis

```
#include <stdlib.h>

void * calloc (size_t cnt, size_t size)
```

Description

The **calloc()** function attempts to obtain a contiguous block of dynamic memory which will hold **cnt** objects, each of length **size**. The block is filled with zeros.

Example

```
#include <stdlib.h>
#include <stdio.h>

struct test {
    int a[20];
} * ptr;

    /* Allocate space for 20 structures. */

void
main (void)
{

    ptr = calloc(20, sizeof(struct test));
    if(!ptr)
        printf("Failed\n");
    else
        free(ptr);
}
```

See Also

brk(), sbrk(), malloc(), free()

Return Value

A pointer to the block is returned, or zero if the memory could not be allocated.

CEIL

Synopsis

```
#include <math.h>

double ceil (double f)
```

Description

This routine returns the smallest whole number not less than **f**.

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    double j;

    scanf("%lf", &j);
    printf("The ceiling of %lf is %lf\n", j, ceil(j));
}
```

CGETS

Synopsis

```
#include <conio.h>

char * cgets (char * s)
```

Description

The **cgets()** function will read one line of input from the console into the buffer passed as an argument. It does so by repeated calls to `getche()`. As characters are read, they are buffered, with *backspace* deleting the previously typed character, and *ctrl-U* deleting the entire line typed so far. Other characters are placed in the buffer, with a *carriage return* or *line feed (newline)* terminating the function. The collected string is null terminated.

Example

```
#include <conio.h>
#include <string.h>

char buffer[80];

void
main (void)
{
    for(;;) {
        cgets(buffer);
        if(strcmp(buffer, "exit") == 0)
            break;
        cputs("Type 'exit' to finish\n");
    }
}
```

See Also

`getch(), getche(), putch(), cputs()`

Return Value

The return value is the character pointer passed as the sole argument.

COS

Synopsis

```
#include <math.h>

double cos (double f)
```

Description

This function yields the cosine of its argument, which is an angle in radians. The cosine is calculated by expansion of a polynomial series approximation.

Example

```
#include <math.h>
#include <stdio.h>

#define C 3.141592/180.0

void
main (void)
{
    double i;

    for(i = 0 ; i <= 180.0 ; i += 10)
        printf("sin(%3.0f) = %f, cos = %f\n", i, sin(i*C), cos(i*C));
}
```

See Also

sin(), tan(), asin(), acos(), atan(), atan2()

Return Value

A double in the range -1 to +1.

COSH, SINH, TANH

Synopsis

```
#include <math.h>

double cosh (double f)
double sinh (double f)
double tanh (double f)
```

Description

These functions are the hyperbolic implementations of the trigonometric functions; `cos()`, `sin()` and `tan()`.

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    printf("%f\n", cosh(1.5));
    printf("%f\n", sinh(1.5));
    printf("%f\n", tanh(1.5));
}
```

Return Value

The function **`cosh()`** returns the hyperbolic cosine value.
The function **`sinh()`** returns the hyperbolic sine value.
The function **`tanh()`** returns the hyperbolic tangent value.

CPUTS

Synopsis

```
#include <conio.h>

void cputs (const char * s)
```

Description

The **cputs()** function writes its argument string to the console, outputting *carriage returns* before each *newline* in the string. It calls `putch()` repeatedly. On a hosted system **cputs()** differs from `puts()` in that it reads the console directly, rather than using file I/O. In an embedded system **cputs()** and `puts()` are equivalent.

Example

```
#include <conio.h>
#include <string.h>

char buffer[80];

void
main (void)
{
    for(;;) {
        cgets(buffer);
        if(strcmp(buffer, "exit") == 0)
            break;
        cputs("Type 'exit' to finish\n");
    }
}
```

See Also

`cputs(), puts(), putch()`

CTIME

Synopsis

```
#include <time.h>

char * ctime (time_t * t)
```

Description

The **ctime()** function converts the time in seconds pointed to by its argument to a string of the same form as described for `asctime()`. Thus the example program prints the current time and date.

Example

```
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t clock;

    time(&clock);
    printf("%s", ctime(&clock));
}
```

See Also

`gmtime(), localtime(), asctime(), time()`

Return Value

A pointer to the string.

Note

The example will require the user to provide the `time()` routine as one cannot be supplied with the compiler. See `time()` for more detail.

Data Types

```
typedef long time_t;
```

DI, EI

Synopsis

```
#include <intrpt.h>

void ei (void)
void di (void)
```

Description

The **ei()** and **di()** routines enable and disable interrupts respectively. These are implemented as macros defined in **intrpt.h**. On most processors they will expand to an in-line assembler instruction that sets or clears the interrupt enable or mask bit.

The example shows the use of **ei()** and **di()** around access to a long variable that is modified during an interrupt. If this was not done, it would be possible to return an incorrect value, if the interrupt occurred between accesses to successive words of the count value.

Example

```
#include <intrpt.h>

long count;

void
interrupt tick (void)
{
    count++;
}

long
getticks (void)
{
    long val;    /* Disable interrupts around access
        to count, to ensure consistency.*/
    di();
    val = count;
    ei();
    return val;
}
```

DIV

Synopsis

```
#include <stdlib.h>

div_t div (int numer, int demon)
```

Description

The **div()** function computes the quotient and remainder of the numerator divided by the denominator.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    div_t x;

    x = div(12345, 66);
    printf("quotient = %d, remainder = %d\n", x.quot, x.rem);
}
```

Return Value

Returns the quotient and remainder into the **div_t** structure.

Data Types

```
typedef struct
{
     int quot;
     int rem;
}  div_t;
```

EVAL_POLY

Synopsis

```
#include <math.h>

double eval_poly (double x, const double * d, int n)
```

Description

The **eval_poly()** function evaluates a polynomial, whose coefficients are contained in the array **d**, at **x**, for example:

y = x*x*d2 + x*d1 + d0.

The order of the polynomial is passed in **n**.

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    double x, y;
    double d[3] = {1.1, 3.5, 2.7};

    x = 2.2;
    y = eval_poly(x, d, 2);
    printf("The polynomial evaluated at %f is %f\n", x, y);
}
```

Return Value

A double value, being the polynomial evaluated at **x**.

EXP

Synopsis

```
#include <math.h>

double exp (double f)
```

Description

The **exp()** routine returns the exponential function of its argument, i.e. e to the power of **f**.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double f;

    for(f = 0.0 ; f <= 5 ; f += 1.0)
        printf("e to %1.0f = %f\n", f, exp(f));
}
```

See Also

log(), log10(), pow()

FABS

Synopsis

```
#include <math.h>

double fabs (double f)
```

Description

This routine returns the absolute value of its double argument.

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    printf("%f %f\n", fabs(1.5), fabs(-1.5));
}
```

See Also

abs()

FLOOR

Synopsis

```
#include <math.h>

double floor (double f)
```

Description

This routine returns the largest whole number not greater than **f**.

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    printf("%f\n", floor( 1.5 ));
    printf("%f\n", floor( -1.5));
}
```

FREE

Synopsis

```
#include <stdlib.h>

void free (void * ptr)
```

Description

The **`free()`** function deallocates the block of memory at **`ptr`**, which must have been obtained from a call to `malloc()` or `calloc()`.

Example

```
#include <stdlib.h>
#include <stdio.h>

struct test {
    int a[20];
} * ptr;

        /* Allocate space for 20 structures. */
void
main (void)
{
    ptr = calloc(20, sizeof(struct test));
    if(!ptr)
        printf("Failed\n");
    else
        free(ptr);
}
```

See Also

`malloc(), calloc()`

FREXP

Synopsis

```
#include <math.h>

double frexp (double f, int * p)
```

Description

The **frexp()** function breaks a floating point number into a normalized fraction and an integral power of 2. The integer is stored into the **int** object pointed to by **p**. Its return value x is in the interval (0.5, 1.0) or zero, and **f** equals x times 2 raised to the power stored in ***p**. If **f** is zero, both parts of the result are zero.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double f;
    int i;

    f = frexp(23456.34, &i);
    printf("23456.34 = %f * 2^%d\n", f, i);
}
```

See Also

ldexp()

GETCH, GETCHE

Synopsis

```
#include <conio.h>

char getch (void)
char getche (void)
```

Description

The **getch()** function reads a single character from the console keyboard and returns it without echoing. The **getche()** function is similar but does echo the character typed.

In an embedded system, the source of characters is defined by the particular routines supplied. By default, the library contains a version of **getch()** that will interface to the Lucifer Debugger. The user should supply an appropriate routine if another source is desired, e.g. a serial port.

The module *getch.c* in the SOURCES directory contains model versions of all the console I/O routines. Other modules may also be supplied, e.g. *ser180.c* has routines for the serial port in a Z180.

Example

```
#include <conio.h>

void
main (void)
{
    char c;

    while((c = getche()) != '\n')
        continue;
}
```

See Also

cgets(), cputs(), ungetch()

GETS

Synopsis

```
#include <stdio.h>

char * gets (char * s)
```

Description

The **gets()** function reads a line from standard input into the buffer at **s**, deleting the *newline* (cf. fgets()). The buffer is null terminated. In an embedded system, **gets()** is equivalent to cgets(), and results in getche() being called repeatedly to get characters. Editing (with *backspace*) is available.

Example

```
#include <stdio.h>

void
main (void)
{
    char buf[80];

    printf("Type a line: ");
    if(gets(buf))
        puts(buf);
}
```

See Also

fgets(), freopen(), puts()

Return Value

It returns its argument, or NULL on end-of-file.

GMTIME

Synopsis

```
#include <time.h>

struct tm * gmtime (time_t * t)
```

Description

This function converts the time pointed to by **t** which is in seconds since 00:00:00 on Jan 1, 1970, into a broken down time stored in a structure as defined in **time.h**. The structure is defined in the 'Data Types' section.

Example

```
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t clock;
    struct tm * tp;

    time(&clock);
    tp = gmtime(&clock);
    printf("It's %d in London\n", tp->tm_year+1900);
}
```

See Also

ctime(), asctime(), time(), localtime()

Return Value

Returns a structure of type **tm**.

Note

The example will require the user to provide the ***time()*** *routine as one cannot be supplied with the compiler. See* ***time()*** *for more detail.*

Data Types

```
typedef long time_t;
struct tm {
     int    tm_sec;
     int    tm_min;
     int    tm_hour;
     int    tm_mday;
     int    tm_mon;
     int    tm_year;
     int    tm_wday;
     int    tm_yday;
     int    tm_isdst;
};
```

ISALNUM, ISALPHA, ISDIGIT, ISLOWER et. al.

Synopsis

```
#include <ctype.h>

int isalnum (char c)
int isalpha (char c)
int isascii (char c)
int iscntrl (char c)
int isdigit (char c)
int islower (char c)
int isprint (char c)
int isgraph (char c)
int ispunct (char c)
int isspace (char c)
int isupper (char c)
int isxdigit(char c)
```

Description

These macros, defined in **ctype.h**, test the supplied character for membership in one of several overlapping groups of characters. Note that all except **isascii()** are defined for **c**, if **isascii(c)** is true or if c = EOF.

isalnum (c)	c is in 0-9 or a-z or A-Z
isalpha (c)	c is in A-Z or a-z
isascii (c)	c is a 7 bit ascii character
iscntrl (c)	c is a control character
isdigit (c)	c is a decimal digit
islower (c)	c is in a-z
isprint (c)	c is a printing char
isgraph (c)	c is a non-space printable character
ispunct (c)	c is not alphanumeric
isspace (c)	c is a space, tab or newline
isupper (c)	c is in A-Z
isxdigit (c)	c is in 0-9 or a-f or A-F

Example

```
#include <ctype.h>
#include <stdio.h>
```

```
void
main (void)
{
    char buf[80];
    int i;

    gets(buf);
    i = 0;
    while(isalnum(buf[i]))
        i++;
    buf[i] = 0;
    printf("'%s' is the word\n", buf);
}
```

See Also

toupper(), tolower(), toascii()

KBHIT

Synopsis

```
#include <conio.h>

int kbhit (void)
```

Description

This function returns 1 if a character has been pressed on the console keyboard, 0 otherwise. Normally the character would then be read via `getch()`.

Example

```
#include <conio.h>

void
main (void)
{
    int i;

    while(!kbhit()) {
        cputs("I'm waiting..");
        for(i = 0 ; i != 1000 ; i++)
            continue;
    }
}
```

See Also

`getch(), getche()`

Return Value

Returns one if a character has been pressed on the console keyboard, zero otherwise.

LDEXP

Synopsis

```
#include <math.h>

double ldexp (double f, int i)
```

Description

The **ldexp()** function performs the inverse of `frexp()` operation; the integer **i** is added to the exponent of the floating point **f** and the resultant returned.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double f;

    f = ldexp(1.0, 10);
    printf("1.0 * 2^10 = %f\n", f);
}
```

See Also

`frexp()`

Return Value

The return value is the integer **i** added to the exponent of the floating point value **f**.

LDIV

Synopsis

```
#include <stdlib.h>

ldiv_t ldiv (long number, long denom)
```

Description

The **ldiv()** routine divides the numerator by the denominator, computing the quotient and the remainder. The sign of the quotient is the same as that of the mathematical quotient. Its absolute value is the largest integer which is less than the absolute value of the mathematical quotient.

The **ldiv()** function is similar to the `div()` function, the difference being that the arguments and the members of the returned structure are all of type **long int**.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    ldiv_t lt;

    lt = ldiv(1234567, 12345);
    printf("Quotient = %ld, remainder = %ld\n", lt.quot, lt.rem);
}
```

See Also

`div()`

Return Value

Returns a structure of type **ldiv_t**

Data Types

```
typedef struct {
     long    quot;  /* quotient */
     long    rem;   /* remainder */
} ldiv_t;
```

LOCALTIME

Synopsis

```
#include <time.h>

struct tm * localtime (time_t * t)
```

Description

The **localtime()** function converts the time pointed to by **t** which is in seconds since 00:00:00 on Jan 1, 1970, into a broken down time stored in a structure as defined in **time.h**. The routine **localtime()** takes into account the contents of the global integer time_zone. This should contain the number of minutes that the local time zone is *westward* of Greenwich. Since there is no way under MS-DOS of actually predetermining this value, by default **localtime()** will return the same result as **gmtime()**.

Example

```
#include <stdio.h>
#include <time.h>

char * wday[] = {
    "Sunday", "Monday", "Tuesday", "Wednesday",
    "Thursday", "Friday", "Saturday"
};

void
main (void)
{
    time_t clock;
    struct tm * tp;

    time(&clock);
    tp = localtime(&clock);
    printf("Today is %s\n", wday[tp->tm_wday]);
}
```

See Also

ctime(), asctime(), time()

Return Value

Returns a structure of type **tm**.

Note

The example will require the user to provide the `time()` *routine as one cannot be supplied with the compiler. See* `time()` *for more detail.*

Data Types

```
typedef long time_t;
struct tm {
     int    tm_sec;
     int    tm_min;
     int    tm_hour;
     int    tm_mday;
     int    tm_mon;
     int    tm_year;
     int    tm_wday;
     int    tm_yday;
     int    tm_isdst;
};
```

LOG, LOG10

Synopsis

```
#include <math.h>

double log (double f)
double log10 (double f)
```

Description

The **log()** function returns the natural logarithm of **f**. The function **log10()** returns the logarithm to base 10 of **f**.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double f;

    for(f = 1.0 ; f <= 10.0 ; f += 1.0)
        printf("log(%1.0f) = %f\n", f, log(f));
}
```

See Also

exp(), pow()

Return Value

Zero if the argument is negative.

LONGJMP

Synopsis

```
#include <setjmp.h>

void longjmp (jmp_buf buf, int val)
```

Description

The **longjmp()** function, in conjunction with setjmp(), provides a mechanism for non-local goto's. To use this facility, setjmp() should be called with a **jmp_buf** argument in some outer level function. The call from setjmp() will return 0.

To return to this level of execution, **lonjmp()** may be called with the same **jmp_buf** argument from an inner level of execution. ***Note*** however that the function which called setjmp() must still be active when **longjmp()** is called. Breach of this rule will cause disaster, due to the use of a stack containing invalid data. The **val** argument to **longjmp()** will be the value apparently returned from the setjmp(). This should normally be non-zero, to distinguish it from the genuine setjmp() call.

Example

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

jmp_buf jb;

void
inner (void)
{
    longjmp(jb, 5);
}

void
main (void)
{
    int i;

    if(i = setjmp(jb)) {
        printf("setjmp returned %d\n", i);
        exit(0);
    }
    printf("setjmp returned 0 - good\n");
    printf("calling inner...\n");
```

```
    inner();
    printf("inner returned - bad!\n");
}
```

See Also

`setjmp()`

Return Value

The **`longjmp()`** routine never returns.

Note

*The function which called `setjmp()` must still be active when **`longjmp()`** is called. Breach of this rule will cause disaster, due to the use of a stack containing invalid data.*

MALLOC

Synopsis

```
#include <stdlib.h>

void * malloc (size_t cnt)
```

Description

The **`malloc()`** function attempts to allocate **`cnt`** bytes of memory from the "heap", the dynamic memory allocation area. If successful, it returns a pointer to the block, otherwise zero is returned. The memory so allocated may be freed with `free()`, or changed in size via `realloc()`. The **`malloc()`** routine calls `sbrk()` to obtain memory, and is in turn called by `calloc()`. The **`malloc()`** function does not clear the memory it obtains, unlike `calloc()`.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>


void
main (void)
{
    char * cp;

    cp = malloc(80);
    if(!cp)
        printf("Malloc failed\n");
    else {
        strcpy(cp, "a string");
        printf("block = '%s'\n", cp);
        free(cp);
    }
}
```

See Also

`calloc(), free(), realloc()`

Return Value

A pointer to the memory if it succeeded; `NULL` otherwise.

MEMCHR

Synopsis

```
#include <string.h>

void * memchr (const void * block, int val, size_t length)
```

Description

The **memchr()** function is similar to `strchr()` except that instead of searching null terminated strings, it searches a block of memory specified by length for a particular byte. Its arguments are a pointer to the memory to be searched, the value of the byte to be searched for, and the length of the block. A pointer to the first occurrence of that byte in the block is returned.

Example

```
#include <string.h>
#include <stdio.h>

unsigned int ary[] = {1, 5, 0x6789, 0x23};

void
main (void)
{
    char * cp;

    cp = memchr(ary, 0x89, sizeof ary);
    if(!cp)
        printf("not found\n");
    else
        printf("Found at offset %u\n", cp - (char *)ary);
}
```

See Also

`strchr()`

Return Value

A pointer to the first byte matching the argument if one exists; `NULL` otherwise.

MEMCMP

Synopsis

```
#include <string.h>

int memcmp (const void * s1, const void * s2, size_t n)
```

Description

The **memcmp()** function compares two blocks of memory, of length **n**, and returns a signed value similar to `strncmp()`. Unlike `strncmp()` the comparison does not stop on a null character. The ASCII collating sequence is used for the comparison, but the effect of including non-ASCII characters in the memory blocks on the sense of the return value is indeterminate. Testing for equality is always reliable.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    int buf[10], cow[10], i;

    buf[0] = 1;
    buf[2] = 4;
    cow[0] = 1;
    cow[2] = 5;
    buf[1] = 3;
    cow[1] = 3;
    i = memcmp(buf, cow, 3*sizeof(int));
    if(i < 0)
        printf("less than\n");
    else if(i > 0)
        printf("Greater than\n");
    else
        printf("Equal\n");
}
```

See Also

`strncpy(), strncmp(), strchr(), memset(), memchr()`

Return Value

Returns negative one, zero or one, depending on whether **s1** points to string which is less than, equal to or greater than the string pointed to by **s2** in the collating sequence.

MEMCPY

Synopsis

```
#include <string.h>

void * memcpy (void * d, const void * s, size_t n)
```

Description

The **memcpy()** function copies **n** bytes of memory starting from the location pointed to by **s** to the block of memory pointed to by **d**. The result of copying overlapping blocks is undefined. The **memcpy()** function differs from strcpy() in that it copies a specified number of bytes, rather than all bytes up to a null terminator.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buf[80];

    memset(buf, 0, sizeof buf);
    memcpy(buf, "a partial string", 10);
    printf("buf = '%s'\n", buf);
}
```

See Also

strncpy(), strncmp(), strchr(), memset()

Return Value

The **memcpy()** routine returns its first argument.

MEMMOVE

Synopsis

```
#include <string.h>

void * memmove (void * s1, const void * s2, size_t n)
```

Description

The **memmove()** function is similar to the function memcpy() except copying of overlapping blocks is handled correctly. That is, it will copy forwards or backwards as appropriate to correctly copy one block to another that overlaps it.

See Also

strncpy(), strncmp(), strchr(), memcpy()

Return Value

The function **memmove()** returns its first argument.

MEMSET

Synopsis

```
#include <string.h>

void * memset (void * s, int c, size_t n)
```

Description

The **memset()** function fills **n** bytes of memory starting at the location pointed to by **s** with the byte **c**.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char abuf[20];

    strcpy(abuf, "This is a string");
    memset(abuf, 'x', 5);
    printf("buf = '%s'\n", abuf);
}
```

See Also

strncpy(), strncmp(), strchr(), memcpy(), memchr()

MODF

Synopsis

```
#include <math.h>

double modf (double value, double * iptr)
```

Description

The **modf()** function splits the argument **value** into integral and fractional parts, each having the same sign as **value**. For example, -3.17 would be split into the intergral part (-3) and the fractional part (-0.17).

The integral part is stored as a double in the object pointed to by **iptr**.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double i_val, f_val;

    f_val = modf( -3.17, &i_val);
}
```

Return Value

The signed fractional part of **value**.

PERSIST_CHECK, PERSIST_VALIDATE

Synopsis

```
#include <sys.h>

int persist_check (int flag)
void persist_validate (void)
```

Description

The **persist_check()** function is used with non-volatile RAM variables, declared with the persistent qualifier. It tests the nvram area, using a magic number stored in a hidden variable by a previous call to **persist_validate()** and a checksum also calculated by **persist_validate()**. If the magic number and checksum are correct, it returns true (non-zero). If either are incorrect, it returns zero. In this case it will optionally zero out and re-validate the non-volatile RAM area (by calling **persist_validate()**). This is done if the flag argument is true.

The **persist_validate()** routine should be called after each change to a persistent variable. It will set up the magic number and recalculate the checksum.

Example

```
#include <sys.h>
#include <stdio.h>

persistent long reset_count;

void
main (void)
{
    if(!persist_check(1))
        printf("Reset count invalid - zeroed\n");
    else
        printf("Reset number %ld\n", reset_count);
    reset_count++;          /* update count */
    persist_validate();     /* and checksum */
    for(;;)
        continue;       /* sleep until next reset */
}
```

Return Value

FALSE (zero) if the NV-RAM area is invalid; TRUE (non-zero) if the NVRAM area is valid.

POW

Synopsis

```
#include <math.h>

double pow (double f, double p)
```

Description

The **pow()** function raises its first argument, **f**, to the power **p**.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double f;

    for(f = 1.0 ; f <= 10.0 ; f += 1.0)
        printf("pow(2, %1.0f) = %f\n", f, pow(2, f));
}
```

See Also

log(), log10(), exp()

Return Value

f to the power of **p**.

PRINTF, VPRINTF

Synopsis

```
#include <stdio.h>

int printf (const char * fmt, ...)

#include <stdio.h>
#include <stdarg.h>

int vprintf (const char * fmt, va_list va_arg)
```

Description

The **printf()** function is a formatted output routine, operating on stdout. There are corresponding routines operating on a given stream (fprintf()) or into a string buffer (sprintf()). The **printf()** routine is passed a format string, followed by a list of zero or more arguments. In the format string are conversion specifications, each of which is used to print out one of the argument list values.

Each conversion specification is of the form **%m.nc** where the percent symbol **%** introduces a conversion, followed by an optional width specification **m**. The **n** specification is an optional precision specification (introduced by the dot) and **c** is a letter specifying the type of the conversion.

A minus sign ('-') preceding **m** indicates left rather than right adjustment of the converted value in the field. Where the field width is larger than required for the conversion, blank padding is performed at the left or right as specified. Where right adjustment of a numeric conversion is specified, and the first digit of **m** is 0, then padding will be performed with zeroes rather than blanks. For integer formats, the precision indicates a minimum number of digits to be output, with leading zeros inserted to make up this number if required.

A hash character (**#**) preceding the width indicates that an alternate format is to be used. The nature of the alternate format is discussed below. Not all formats have alternates. In those cases, the presence of the hash character has no effect.

The floating point formats require that the appropriate floating point library is linked. From within HPD this can be forced by selecting the "Float formats in printf" selection in the options menu. From the command line driver, use the option **-LF**.

If the character ***** is used in place of a decimal constant, e.g. in the format **%*d**, then one integer argument will be taken from the list to provide that value. The types of conversion are:

f

Floating point - **m** is the total width and **n** is the number of digits after the decimal point. If **n** is omitted it defaults to 6. If the precision is zero, the decimal point will be omitted unless the alternate format is specified.

e
Print the corresponding argument in scientific notation. Otherwise similar to **f**.

g
Use **e** or **f** format, whichever gives maximum precision in minimum width. Any trailing zeros after the decimal point will be removed, and if no digits remain after the decimal point, it will also be removed.

o x X u d
Integer conversion - in radices 8, 16, 16, 10 and 10 respectively. The conversion is signed in the case of **d**, unsigned otherwise. The precision value is the total number of digits to print, and may be used to force leading zeroes. E.g. **%8.4x** will print at least 4 hex digits in an 8 wide field. Preceding the key letter with an **l** indicates that the value argument is a long integer. The letter **X** prints out hexadecimal numbers using the upper case letters *A-F* rather than *a-f* as would be printed when using **x**. When the alternate format is specified, a leading zero will be supplied for the octal format, and a leading `0x` or `0X` for the hex format.

s
Print a string - the value argument is assumed to be a character pointer. At most **n** characters from the string will be printed, in a field **m** characters wide.

c
The argument is assumed to be a single character and is printed literally.

Any other characters used as conversion specifications will be printed. Thus **%%** will produce a single percent sign.

The **vprintf()** function is similar to **printf()** but takes a variable argument list pointer rather than a list of arguments. See the description of `va_start()` for more information on variable argument lists. An example of using **vprintf()** is given below.

Example

```
printf("Total = %4d%%", 23)
       yields 'Total =   23%'

printf("Size is %lx" , size)
       where size is a long, prints size
       as hexadecimal.

printf("Name = %.8s", "a1234567890")
       yields 'Name = a1234567'

printf("xx%*d", 3, 4)
       yields 'xx  4'
```

```
/* vprintf example */

#include        <stdio.h>

int
error (char * s, ...)
{
    va_list ap;

    va_start(ap, s);
    printf("Error: ");
    vprintf(s, ap);
    putchar('\n');
    va_end(ap);
}

void
main (void)
{
    int i;

    i = 3;
    error("testing 1 2 %d", i);
}
```

See Also

`fprintf(), sprintf()`

Return Value

The **`printf()`** and **`vprintf()`** functions return the number of characters written to `stdout`.

PUTCH

Synopsis

```
#include <conio.h>

void putch (char c)
```

Description

The **putch()** function outputs the character **c** to the console screen, prepending a *carriage return* if the character is a *newline*. In a CP/M or MS-DOS system this will use one of the system I/O calls. In an embedded system this routine, and associated others, will be defined in a hardware dependent way. The standard **putch()** routines in the embedded library interface either to a serial port or to the Lucifer Debugger.

Example

```
#include <conio.h>

char * x = "This is a string";

void
main (void)
{
    char * cp;

    cp = x;
    while(*x)
        putch(*x++);
    putch('\n');
}
```

See Also

cgets(), cputs(), getch(), getche()

PUTS

Synopsis

```
#include <stdio.h>

int puts (const char * s)
```

Description

The **puts()** function writes the string **s** to the *stdout stream*, appending a *newline*. The null character terminating the string is not copied.

Example

```
#include <stdio.h>

void
main (void)
{
    puts("Hello, world!");
}
```

See Also

fputs(), gets(), freopen(), fclose()

Return Value

EOF is returned on error; zero otherwise.

QSORT

Synopsis

```
#include <stdlib.h>

void qsort (void * base, size_t nel, size_t width,
          int (*func)(const void *, const void *))
```

Description

The **qsort()** function is an implementation of the quicksort algorithm. It sorts an array of **nel** items, each of length **width** bytes, located contiguously in memory at **base**. The argument **func** is a pointer to a function used by **qsort()** to compare items. It calls **func** with pointers to two items to be compared. If the first item is considered to be greater than, equal to or less than the second then **func** should return a value greater than zero, equal to zero or less than zero respectively.

Example

```
#include <stdio.h>
#include <stdlib.h>

int aray[] = {
    567, 23, 456, 1024, 17, 567, 66
};

int
sortem (const void * p1, const void * p2)
{
    return *(int *)p1 - *(int *)p2;
}

void
main (void)
{
    register int i;

    qsort(aray, sizeof aray/sizeof aray[0], sizeof aray[0], sortem);
    for(i = 0 ; i != sizeof aray/sizeof aray[0] ; i++)
        printf("%d\t", aray[i]);
    putchar('\n');
}
```

Note

The function parameter must be a pointer to a function of type similar to:

```
int func (const void *, const void *)
```

*i.e. it must accept two const void * parameters, and must be prototyped.*

RAM_VECTOR, CHANGE_VECTOR, READ_RAM_VECTOR

Synopsis

```
#include <intrpt.h>

void RAM_VECTOR (unsigned vector, isr func, unsigned psw)
void CHANGE_VECTOR (unsigned vector, isr func)
void (* READ_RAM_VECTOR (unsigned vector)(void))
```

Description

The **RAM_VECTOR()**, **CHANGE_VECTOR()** and **READ_RAM_VECTOR()** macros are used to initialize, modify and read interrupt vectors which are directed through internal RAM based interrupt vectors. These macros should only be used for vectors which need to be modifiable, so as to point at different interrupt functions at different points in the program. The **CHANGE_VECTOR()** and **READ_RAM_VECTOR()** macros should only be used with interrupt vectors which have been initialized using **RAM_VECTOR()**, otherwise garbage will be returned.

Please refer to the section "*Interrupt Handling in C*" in this manual for further details.

Example

```
volatile unsigned char wait_flag;

interrupt void wait_handler(void)
{
    ++wait_flag;
}

void wait_for_serial_intr(void)
{
    interrupt void (*old_handler)(void);

    di();
    old_handler = READ_RAM_VECTOR(RXI);
    wait_flag = 0;
    CHANGE_VECTOR(RXI, wait_handler);
```

See Also

di(), ei(), ROM_VECTOR()

RAND

Synopsis

```
#include <stdlib.h>

int rand (void)
```

Description

The **rand()** function is a pseudo-random number generator. It returns an integer in the range 0 to 32767, which changes in a pseudo-random fashion on each call. The algorithm will produce a deterministic sequence if started from the same point. The starting point is set using the **srand()** call. The example shows use of the **time()** function to generate a different starting point for the sequence each time.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t toc;
    int i;

    time(&toc);
    srand((int)toc);
    for(i = 0 ; i != 10 ; i++)
        printf("%d\t", rand());
    putchar('\n');
}
```

See Also

srand()

Note

The example will require the user to provide the `time()` *routine as one cannot be supplied with the compiler. See* `time()` *for more detail.*

REALLOC

Synopsis

```
#include <stdlib.h>

void * realloc (void * ptr, size_t cnt)
```

Description

The **realloc()** function frees the block of memory at **ptr**, which should have been obtained by a previous call to malloc(), calloc() or **realloc()**, then attempts to allocate **cnt** bytes of dynamic memory, and if successful copies the contents of the block of memory located at **ptr** into the new block.

At most, **realloc()** will copy the number of bytes which were in the old block, but if the new block is smaller, will only copy **cnt** bytes.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void
main (void)
{
    char * cp;

    cp = malloc(255);
    if(gets(cp))
        cp = realloc(cp, strlen(cp)+1);
    printf("buffer now %d bytes long\n", strlen(cp)+1);
}
```

See Also

malloc(), calloc()

Return Value

A pointer to the new (or resized) block. NULL if the block could not be expanded. A request to shrink a block will never fail.

ROM_VECTOR

Synopsis

```
#include <intrpt.h>

void ROM_VECTOR (unsigned vector, isr func, unsigned psw)
```

Description

The **ROM_VECTOR()** macro is used to set up a "*hard coded*" ROM vector, which points to an interrupt handler. This macro does not generate any code which is executed at run-time, so it can be placed anywhere in your code. **ROM_VECTOR()** generates in-line assembler code, so the vector address passed to it may be in any format acceptable to the assembler.

Please refer to the section "*Interrupt Handling in C*", in this manual for further details.

See Also

`di(), ei(), RAM_VECTOR()`

SCANF, VSCANF

Synopsis

```
#include <stdio.h>

int scanf (const char * fmt, ...)

#include <stdio.h>
#include <stdarg.h>

int vscanf (const char *, va_list ap)
```

Description

The **scanf()** function performs formatted input ("de-editing") from the *stdin stream*. Similar functions are available for streams in general, and for strings. The function **vscanf()** is similar, but takes a pointer to an argument list rather than a series of additional arguments. This pointer should have been initialised with `va_start()`.

The input conversions are performed according to the **fmt** string; in general a character in the format string must match a character in the input; however a space character in the format string will match zero or more "white space" characters in the input, i.e. *spaces, tabs or newlines*.

A conversion specification takes the form of the character **%**, optionally followed by an assignment suppression character ('*****'), optionally followed by a numerical maximum field width, followed by a conversion specification character. Each conversion specification, unless it incorporates the assignment suppression character, will assign a value to the variable pointed at by the next argument. Thus if there are two conversion specifications in the **fmt** string, there should be two additional pointer arguments.

The conversion characters are as follows:

o x d
Skip white space, then convert a number in base 8, 16 or 10 radix respectively. If a field width was supplied, take at most that many characters from the input. A leading minus sign will be recognized.

f
Skip white space, then convert a floating number in either conventional or scientific notation. The field width applies as above.

s
Skip white space, then copy a maximal length sequence of non-white-space characters. The pointer argument must be a pointer to char. The field width will limit the number of characters copied. The resultant string will be null terminated.

c
Copy the next character from the input. The pointer argument is assumed to be a pointer to char. If a field

width is specified, then copy that many characters. This differs from the **s** format in that white space does not terminate the character sequence.

The conversion characters **o**, **x**, **u**, **d** and **f** may be preceded by an **l** to indicate that the corresponding pointer argument is a pointer to long or double as appropriate. A preceding **h** will indicate that the pointer argument is a pointer to short rather than int.

Example

```
scanf("%d %s", &a, &c)
     with input " 12s"
     will assign 12 to a, and "s" to s.

scanf("%3cd %lf", &c, &f)
     with input " abcd -3.5"
     will assign " abc" to c, and -3.5 to f.
```

See Also

`fscanf(), sscanf(), printf(), va_arg()`

Return Value

The **`scanf()`** function returns the number of successful conversions; `EOF` is returned if end-of-file was seen before any conversions were performed.

SETJMP

Synopsis

```
#include <setjmp.h>

int setjmp (jmp_buf buf)
```

Description

The **setjmp()** function is used with longjmp() for non-local goto's. See longjmp() for further information.

Example

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

jmp_buf jb;

void
inner (void)
{
    longjmp(jb, 5);
}

void
main (void)
{
    int i;

    if(i = setjmp(jb)) {
        printf("setjmp returned %d\n", i);
        exit(0);
    }
    printf("setjmp returned 0 - good\n");
    printf("calling inner...\n");
    inner();
    printf("inner returned - bad!\n");
}
```

See Also

longjmp()

Return Value

The **`setjmp()`** function returns zero after the real call, and non-zero if it apparently returns after a call to `longjmp()`.

SET_VECTOR

Synopsis

```
#include <intrpt.h>

isr set_vector (isr * vector, isr func)
```

Description

This routine allows an interrupt vector to be initialized. The first argument should be the address of the interrupt vector (not the vector number but the actual address) cast to a pointer to **isr**, which is a typedef'd pointer to an interrupt function. The second argument should be the function which you want the interrupt vector to point to. This must be declared using the **interrupt** type qualifier.

Not all compilers support this routine; the macros ROM_VECTOR(), RAM_VECTOR() and CHANGE_VECTOR() are used with some processors. These routines are to be preferred even where **set_vector()** is supported. See **intrpt.h** or the processor specific manual section to determine what is supported for a particular compiler.

The example shown sets up a vector for the DOS ctrl-BREAK interrupt.

Example

```
#include <signal.h>
#include <stdlib.h>
#include <intrpt.h>

static far interrupt void
brkintr (void)
{
    exit(-1);
}

#define BRKINT  0x23
#define BRKINTV ((far isr *)(BRKINT * 4))

void
set_trap (void)
{
    set_vector(BRKINTV, brkintr);
}
```

See Also

di(), ei(), ROM_VECTOR(), RAM_VECTOR(), CHANGE_VECTOR()

Return Value

The return value of **set_vector()** is the previous contents of the vector, if **set_vector()** is implemented as a function. If it is implemented as a macro, it has no return value.

Note

*The **set_vector()** routine is equivalent to ROM_VECTOR() and is present only for compatibility with version 5 and 6 HI-TECH compilers. It is suggested that ROM_VECTOR() be used in place of **set_vector()** for maximum compatibility with future versions of HI-TECH C.*

Data Types

```
typedef interrupt void (*isr)(void)
```

SIN

Synopsis

```
#include <math.h>

double sin (double f)
```

Description

This function returns the sine function of its argument.

Example

```
#include <math.h>
#include <stdio.h>

#define C 3.141592/180.0

void
main (void)
{
    double i;

    for(i = 0 ; i <= 180.0 ; i += 10)
        printf("sin(%3.0f) = %f, cos = %f\n", i, sin(i*C), cos(i*C));
}
```

See Also

`cos(), tan(), asin(), acos(), atan(), atan2()`

Return Value

Sine vale of **f**.

SPRINTF, VSPRINTF

Synopsis

```
#include <stdio.h>

int sprintf (char * buf, const char * fmt, ...)

#include <stdio.h>
#include <stdarg.h>

int vsprintf (char * buf, const char * fmt, va_list ap)
```

Description

The **sprintf()** function operates in a similar fashion to printf(), except that instead of placing the converted output on the *stdout stream*, the characters are placed in the buffer at **buf**. The resultant string will be null terminated, and the number of characters in the buffer will be returned.

The **vsprintf()** function is similar to **sprintf()** but takes a variable argument list pointer rather than a list of arguments. See the description of va_start() for more information on variable argument lists.

See Also

printf(), fprintf(), sscanf()

Return Value

Both these routines return the number of characters placed into the buffer.

SQRT

Synopsis

```
#include <math.h>

double sqrt (double f)
```

Description

The function **sqrt()**, implements a square root routine using Newton's approximation.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double i;

    for(i = 0 ; i <= 20.0 ; i += 1.0)
        printf("square root of %.1f = %f\n", i, sqrt(i));
}
```

See Also

exp()

Return Value

Returns the value of the square root.

Note

A domain error occurs if the argument is negative.

SRAND

Synopsis

```
#include <stdlib.h>

void srand (unsigned int seed)
```

Description

The **srand()** function initializes the random number generator accessed by `rand()` with the given **seed**. This provides a mechanism for varying the starting point of the pseudo-random sequence yielded by `rand()`. On the z80, a good place to get a truly random seed is from the refresh register. Otherwise timing a response from the console will do, or just using the system time.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t toc;
    int i;

    time(&toc);
    srand((int)toc);
    for(i = 0 ; i != 10 ; i++)
        printf("%d\t", rand());
    putchar('\n');
}
```

See Also

`rand()`

SSCANF, VSSCANF

Synopsis

```
#include <stdio.h>

int sscanf (const char * buf, const char * fmt, ...)

#include <stdio.h>
#include <stdarg.h>

int vsscanf (const char * buf, const char * fmt, va_list ap)
```

Description

The **sscanf()** function operates in a similar manner to `scanf()`, except that instead of the conversions being taken from `stdin`, they are taken from the string at **buf**.

The **vsscanf()** function takes an argument pointer rather than a list of arguments. See the description of `va_start()` for more information on variable argument lists.

See Also

`scanf(), fscanf(), sprintf()`

Return Value

Returns the value of `EOF` if an input failure occurs, else returns the number of input items.

STRCAT

Synopsis

```
#include <string.h>

char * strcat (char * s1, const char * s2)
```

Description

This function appends (catenates) string **s2** to the end of string **s1**. The result will be null terminated. The argument **s1** must point to a character array big enough to hold the resultant string.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strcpy(buffer, "Start of line");
    s1 = buffer;
    s2 = " ... end of line";
    strcat(s1, s2);
    printf("Length = %d\n", strlen(buffer));
    printf("string = \"%s\"\n", buffer);
}
```

See Also

strcpy(), strcmp(), strncat(), strlen()

Return Value

The value of **s1** is returned.

STRCHR, STRICHR

Synopsis

```
#include <string.h>

char * strchr (const char * s, int c)
char * strichr (const char * s, int c)
```

Description

The **strchr()** function searches the string **s** for an occurrence of the character **c**. If one is found, a pointer to that character is returned, otherwise NULL is returned.

The **strichr()** function is the case-insensitive version of this function.

Example

```
#include <strings.h>
#include <stdio.h>

void
main (void)
{
    static char temp[] = "Here it is...";
    char c = 's';

    if(strchr(temp, c))
        printf("Character %c was found in string\n", c);
    else
        printf("No character was found in string");
}
```

See Also

strrchr(), strlen(), strcmp()

Return Value

A pointer to the first match found, or NULL if the character does not exist in the string.

Note

Although the function takes an integer argument for the character, only the lower 8 bits of the value are used.

STRCMP, STRICMP

Synopsis

```
#include <string.h>

int strcmp (const char * s1, const char * s2)
int stricmp (const char * s1, const char * s2)
```

Description

The **strcmp()** function compares its two, null terminated, string arguments and returns a signed integer to indicate whether **s1** is less than, equal to or greater than **s2**. The comparison is done with the standard collating sequence, which is that of the ASCII character set.

The **stricmp()** function is the case-insensitive version of this function.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    int i;

    if((i = strcmp("ABC", "ABc")) < 0)
        printf("ABC is less than ABc\n");
    else if(i > 0)
        printf("ABC is greater than ABc\n");
    else
        printf("ABC is equal to ABc\n");
}
```

See Also

strlen(), strncmp(), strcpy(), strcat()

Return Value

A signed integer less than, equal to or greater than zero.

Note

Other C implementations may use a different collating sequence; the return value is negative, zero or positive, i.e. do not test explicitly for negative one (-1) or one (1).

STRCPY

Synopsis

```
#include <string.h>

char * strcpy (char * s1, const char * s2)
```

Description

This function copies a null terminated string **s2** to a character array pointed to by **s1**. The destination array must be large enough to hold the entire string, including the null terminator.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strcpy(buffer, "Start of line");
    s1 = buffer;
    s2 = " ... end of line";
    strcat(s1, s2);
    printf("Length = %d\n", strlen(buffer));
    printf("string = \"%s\"\n", buffer);
}
```

See Also

strncpy(), strlen(), strcat(), strlen()

Return Value

The destination buffer pointer **s1** is returned.

STRCSPN

Synopsis

```
#include <string.h>

size_t strcspn (const char * s1, const char * s2)
```

Description

The **strcspn()** function returns the length of the initial segment of the string pointed to by **s1** which consists of characters NOT from the string pointed to by **s2**.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    static char set[] = "xyz";

    printf("%d\n", strcspn( "abcdevwxyz", set));
    printf("%d\n", strcspn( "xxxbcadefs", set));
    printf("%d\n", strcspn( "1234567890", set));
}
```

See Also

strspn()

Return Value

Returns the length of the segment.

STRDUP

Synopsis

```
#include <string.h>

char * strdup (const char * s1)
```

Description

The **strdup()** function returns a pointer to a new string which is a duplicate of the string pointed to by **s1**. The space for the new string is obtained using malloc(). If the new string cannot be created, a null pointer is returned.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    char * ptr;

    ptr = strdup("This is a copy");
    printf("%s\n", ptr);
}
```

Return Value

Pointer to the new string, or NULL if the new string cannot be created.

STRLEN

Synopsis

```
#include <string.h>

size_t strlen (const char * s)
```

Description

The **strlen()** function returns the number of characters in the string **s**, not including the null terminator.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strcpy(buffer, "Start of line");
    s1 = buffer;
    s2 = " ... end of line";
    strcat(s1, s2);
    printf("Length = %d\n", strlen(buffer));
    printf("string = \"%s\"\n", buffer);
}
```

Return Value

The number of characters preceding the null terminator.

STRNCAT

Synopsis

```
#include <string.h>

char * strncat (char * s1, const char * s2, size_t n)
```

Description

This function appends (catenates) string **s2** to the end of string **s1**. At most **n** characters will be copied, and the result will be null terminated. **s1** must point to a character array big enough to hold the resultant string.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strcpy(buffer, "Start of line");
    s1 = buffer;
    s2 = " ... end of line";
    strncat(s1, s2, 5);
    printf("Length = %d\n", strlen(buffer));
    printf("string = \"%s\"\n", buffer);
}
```

See Also

strcpy(), strcmp(), strcat(), strlen()

Return Value

The value of **s1** is returned.

STRNCMP, STRNICMP

Synopsis

```
#include <string.h>

int strncmp (const char * s1, const char * s2, size_t n)
int strnicmp (const char * s1, const char * s2, size_t n)
```

Description

The **strcmp()** function compares its two, null terminated, string arguments, up to a maximum of **n** characters, and returns a signed integer to indicate whether **s1** is less than, equal to or greater than **s2**. The comparison is done with the standard collating sequence, which is that of the ASCII character set.

The **stricmp()** function is the case-insensitive version of this function.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    int i;

    i = strcmp("abcxyz", "abcxyz");
    if(i == 0)
        printf("Both strings are equal\n");
    else if(i > 0)
        printf("String 2 less than string 1\n");
    else
        printf("String 2 is greater than string 1\n");
}
```

See Also

strlen(), strcmp(), strcpy(), strcat()

Return Value

A signed integer less than, equal to or greater than zero.

Note

Other C implementations may use a different collating sequence; the return value is negative, zero or positive, i.e. do not test explicitly for negative one (-1) or one (1).

STRNCPY

Synopsis

```
#include <string.h>

char * strncpy (char * s1, const char * s2, size_t n)
```

Description

This function copies a null terminated string **s2** to a character array pointed to by **s1**. At most **n** characters are copied. If string **s2** is longer than **n** then the destination string will not be null terminated. The destination array must be large enough to hold the entire string, including the null terminator.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strncpy(buffer, "Start of line", 6);
    s1 = buffer;
    s2 = " ... end of line";
    strcat(s1, s2);
    printf("Length = %d\n", strlen(buffer));
    printf("string = \"%s\"\n", buffer);
}
```

See Also

strcpy(), strcat(), strlen(), strcmp()

Return Value

The destination buffer pointer **s1** is returned.

STRPBRK

Synopsis

```
#include <string.h>

char * strpbrk (const char * s1, const char * s2)
```

Description

The **strpbrk()** function returns a pointer to the first occurrence in string **s1** of any character from string **s2**, or a null pointer if no character from **s2** exists in **s1**.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    char * str = "This is a string.";

    while(str != NULL) {
        printf( "%s\n", str );
        str = strpbrk( str+1, "aeiou" );
    }
}
```

Return Value

Pointer to the first matching character, or NULL if no character found.

STRRCHR, STRRICHR

Synopsis

```
#include <string.h>

char * strrchr (char * s, int c)
char * strrichr (char * s, int c)
```

Description

The **strrchr()** function is similar to the **strchr()** function, but searches from the end of the string rather than the beginning, i.e. it locates the *last* occurrence of the character **c** in the null terminated string **s**. If successful it returns a pointer to that occurrence, otherwise it returns NULL.

The **strrichr()** function is the case-insensitive version of this function.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    char * str = "This is a string.";

    while(str != NULL) {
      printf( "%s\n", str );
      str = strrchr( str+1, 's');
    }
}
```

See Also

strchr(), strlen(), strcmp(), strcpy(), strcat()

Return Value

A pointer to the character, or NULL if none is found.

STRSPN

Synopsis

```
#include <string.h>

size_t strspn (const char * s1, const char * s2)
```

Description

The **strspn()** function returns the length of the initial segment of the string pointed to by **s1** which consists entirely of characters from the string pointed to by **s2**.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    printf("%d\n", strspn("This is a string", "This"));
    printf("%d\n", strspn("This is a string", "this"));
}
```

See Also

strcspn()

Return Value

The length of the segment.

STRSTR, STRISTR

Synopsis

```
#include <string.h>

char * strstr (const char * s1, const char * s2)
char * stristr (const char * s1, const char * s2)
```

Description

The **strstr()** function locates the first occurrence of the sequence of characters in the string pointed to by **s2** in the string pointed to by **s1**.

The **stristr()** routine is the case-insensitive version of this function.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    printf("%d\n", strstr("This is a string", "str"));
}
```

Return Value

Pointer to the located string or a null pointer if the string was not found.

STRTOK

Synopsis

```
#include <string.h>

char * strtok (char * s1, const char * s2)
```

Description

A number of calls to **strtok()** breaks the string **s1** (which consists of a sequence of zero or more text tokens separated by one or more characters from the separator string **s2**) into its separate tokens.

The first call must have the string **s1**. This call returns a pointer to the first character of the first token, or NULL if no tokens were found. The inter-token separator character is overwritten by a null character, which terminates the current token.

For subsequent calls to **strtok()**, **s1** should be set to a null pointer. These calls start searching from the end of the last token found, and again return a pointer to the first character of the next token, or NULL if no further tokens were found.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    char * ptr;
    char * buf = "This is a string of words.";
    char * sep_tok = ".,?! ";

    ptr = strtok(buf, sep_tok);
    while(ptr != NULL) {
        printf("%s\n", ptr);
        ptr = strtok(NULL, sep_tok);
    }
}
```

Return Value

Returns a pointer to the first character of a token, or a null pointer if no token was found.

Note

*The separator string **s2** may be different from call to call.*

TAN

Synopsis

```
#include <math.h>

double tan (double f)
```

Description

The **tan()** function calculates the tangent of **f**.

Example

```
#include <math.h>
#include <stdio.h>

#define C 3.141592/180.0

void
main (void)
{
    double i;

    for(i = 0 ; i <= 180.0 ; i += 10)
        printf("tan(%3.0f) = %f\n", i, tan(i*C));
}
```

See Also

sin(), cos(), asin(), acos(), atan(), atan2()

Return Value

The tangent of **f**.

TIME

Synopsis

```
#include <time.h>

time_t time (time_t * t)
```

Description

This function is not provided as it is dependant on the target system supplying the current time. This function will be user implemented. When implemented, this function should return the current time in seconds since 00:00:00 on Jan 1, 1970. If the argument **t** is not equal to NULL, the same value is stored into the object pointed to by **t**.

Example

```
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t clock;

    time(&clock);
    printf("%s", ctime(&clock));
}
```

See Also

ctime(), gmtime(), localtime(), asctime()

Return Value

This routine when implemented will return the current time in seconds since 00:00:00 on Jan 1, 1970.

Note

*The **time()** routine is not supplied, if required the user will have to implement this routine to the specifications outlined above.*

TOLOWER, TOUPPER, TOASCII

Synopsis

```
#include <ctype.h>

char toupper (int c)
char tolower (int c)
char toascii (int c)
```

Description

The **toupper()** function converts its lower case alphabetic argument to upper case, the **tolower()** routine performs the reverse conversion and the **toascii()** macro returns a result that is guaranteed in the range 0-0177. The functions **toupper()** and **tolower()** return their arguments if it is not an alphabetic character.

Example

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

void
main (void)
{
    char * array1 = "aBcDE";
    int i;

    for(i=0;i < strlen(array1); ++i)  {
        printf("%c", tolower(array1[i]));
    }
    printf("\n");
}
```

See Also

islower(), isupper(), isascii(), et. al.

VA_START, VA_ARG, VA_END

Synopsis

```
#include <stdarg.h>

void va_start (va_list ap, parmN)
type va_arg  (ap, type)
void va_end  (va_list ap)
```

Description

These macros are provided to give access in a portable way to parameters to a function represented in a prototype by the ellipsis symbol (...), where type and number of arguments supplied to the function are not known at compile time.

The rightmost parameter to the function (shown as **parmN**) plays an important role in these macros, as it is the starting point for access to further parameters. In a function taking variable numbers of arguments, a variable of type **va_list** should be declared, then the macro **va_start()** invoked with that variable and the name of **parmN**. This will initialize the variable to allow subsequent calls of the macro **va_arg()** to access successive parameters.

Each call to **va_arg()** requires two arguments; the variable previously defined and a type name which is the type that the next parameter is expected to be. Note that any arguments thus accessed will have been widened by the default conventions to *int*, *unsigned int* or *double*. For example if a character argument has been passed, it should be accessed by **va_arg(ap, int)** since the *char* will have been widened to *int*.

An example is given below of a function taking one integer parameter, followed by a number of other parameters. In this example the function expects the subsequent parameters to be pointers to char, but note that the compiler is not aware of this, and it is the programmers responsibility to ensure that correct arguments are supplied.

Example

```
#include <stdio.h>
#include <stdarg.h>

void
pf (int a, ...)
{
    va_list ap;

    va_start(ap, a);
    while(a--)
        puts(va_arg(ap, char *));
```

```
    va_end(ap);
}

void
main (void)
{
    pf(3, "Line 1", "line 2", "line 3");
}
```

XTOI

Synopsis

```
#include <stdlib.h>

unsigned xtoi (const char * s)
```

Description

The **xtoi()** function scans the character string passed to it, skipping leading blanks reading an optional sign, and converts an ASCII representation of a hexadecimal number to an integer.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[80];
    int i;

    gets(buf);
    i = xtoi(buf);
    printf("Read %s: converted to %x\n", buf, i);
}
```

See Also

atoi()

Return Value

A signed integer. If no number is found in the string, zero will be returned.

Index

Symbols

! macro quote character 176
#asm directive 149
#define 100
#endasm directive 149
#pragma directives 150
#undef 109
$ assembler control character 177
$ assembler label character 164
$ location counter symbol 165
% macro argument prefix 177
. psect address symbol 187
... symbol 133
.as files 83
.c files 83
.cmd files 92, 196
.crf files 99, 161
.hex files 103, 105, 106, 109
.lib files 83, 194, 196
.lnk files 47, 190
.lst files 80, 96, 99
.map files 83
.obj files 83, 187, 196
.omf files 106
.pre files 79
.prj files 82, 87
.pro files 106
.sdb files 27
.sym files 112, 186, 189
.ubr files 109
/ psect address symbol 187
;; macro comment suppresser 174
<<>> menu 57, 70
    About HPDXA 70
    Setup 70
<> macro argument list 176
? assembler special label character 164
??nnnn type symbols 165, 174
?_xxxx type symbols 192
?a_xxxx type symbols 192
@ address construct 132, 135
@ in a macro 173
_ assembler special label character 164
__Bxxxx type symbols 39, 158
__Hxxxx type symbols 37, 158
__Lxxxx type symbols 37, 158
_xxxx type symbols 115

Numerics

32-bit floating point format 122
64-bit floating point format 123
80C51XA data book 168

A

ABS 272
abs psect flag 170
absolute object files 187
absolute psects 170
absolute variables 45, 132, 152
ACOS 273
addresses
    link 182, 187
    link addresses
        load 34

load 182, 187
unresolved in listing file 28
addressing unit size 170
align psect flag 171
aligning labels 171
alignment
psects 171
ANSI standard
conformance 109
disabling 80
implementation-defined behaviour 111
argument area 133
argument passing 133
ASCII table 87
ASCTIME 274
ASIN 276
asm() C directive 149
assembler 27, 161
accessing C objects 148
base specifiers 163
character set 163
command line options 161
conditional 173
controls 177
table of 177
default radix 163
delimiters 164
directives 168
expressions 166
functions 147
generating Avocet AVMAC 109
generating from C 109
identifiers 164, 165
significance of 164
include files 178
initializing
bytes 172
words 172
in-line 148
label field 167
line numbers 163
mixing with C 147
operators, table of 167
optimizer 27
options, table of 162
pseudo-ops 168
radix specifiers 163
repeating macros 174
reserving
bytes 172
locations 172
words 172
SFR names 165
special characters 164
strings 165
symbols 168
user-defined symbols 164
assembler code 163
called by C 147
assembler errors
suppressing 163
assembler files 22
preprocessing 80, 106
assembler listings 28, 80, 96, 162
$ character 177
disabling macro expansion 178
excluding conditional code 178
expanding macros 177
including conditional code 177
new page 177
page length 178
page width 163, 178
restoring flags 178
saving flags 178
title 178
turning off 178

turning on 178
assembler options 161
assembler temporary labels 165
ASSERT 277
ASXA
controls 177
table of 177
directives 168
table of 169
expressions 166
generated symbols 165
location counter 165
numbers and bases 164
operators, table of 167
options 161
table of 162
special characters 164
statement format 167
statements 167
symbols 165
temporary labels 165
ASXA controls
$ character 177
cond 177
eject 177
gen 177
include 178
list 178
nocond 178
nogen 178
nolist 178
noxref 178, 179
pagelength 178
pagewidth 178
restore 178
save 178
table of 177
title 178
xref 179
ASXA directives
db 172
dd 172
df 172
ds 172
dw 172
else 172
elseif 172
end 169
endif 172
endm 173
equ 171
extrn 168
global 168
if 172
irp 175
irpc 175
local 165, 174
macro 173
org 171
psect 168, 169
psect flags
abs 170
align 171
bit 170
class 171
global 170
limit 171
local 170
ovrld 170
pure 170
reloc 171
size 171
space 171
table of 170
public 168
rept 174

set 171
signat 157, 176
table of 169
ASXA options
-C 161
-D 162
-Dsymb 162
-Flength 162
-I 162
-Llistfile 162
-Ooutfile 162
-Q 163
-S 163
table of 162
-U 163
-V 163
-Wwidth 163
-X 163
ATAN 278
ATAN2 279
ATOF 280
ATOI 281
ATOL 282
auto indent mode 64
auto variable area 133
auto variables 131, 207
autoexec.bat 205
auto-repeat rate, mouse 70
autosave 71

B

ballistic threshold, mouse 70
banked
interrupts 141
keyword 141
bases
assembler 164
C source 118
batch files 92, 101
begin block 66
binary constants
assembler 164
C 118
binary files 96, 156
bit 166
addressable SFRs 120
fields 123
keyword 119
psect flag 170
variables 119
bit types 139, 170
block commands 65, 66
begin 66
comment/uncomment 68
copy 68
delete 68
end 66
go to end 66
go to start 66
hide/display 66
indent 68
key equivalents
table of 66
move 68
outdent 68
read 68
write 68
block hide 69
boolean types 118
BSEARCH 283
bss psect 132, 139, 182
clearing 115, 182
button
continue 62
fix 63
help 63

hide 63
next 65
previous 65
search 65
buttons 57

C

C mode 65
C source listings 23, 80
example of 23
calculator 87
call graph 192
CALLOC 285
CEIL 286
CGETS 287
CHANGE_VECTOR 331
CHANGE_VECTOR macro 144
char types 121
signed 110
unsigned 110
chicken sheds 165
class psect flag 171
classes 185
address ranges 184
boundary argument 189
upper address limit 189
clear clipboard 70
clearing
bss psect 115
rbss psect 115
clipboard 65, 69
clear 73
copy 72
cut 72
delete selection 73
hide 73
paste 73
selecting text 69
show 73
clipboard commands 69
clear 70
copy 68, 69
cut 69
delete 68, 70
hide 70
paste 70
show 70
clist utility 23
clock, enabling 57
clutches 44
code
at specific addresses 155
keyword 125, 127
optimizations 158
pointers 130
qualifier 155
code generator 26
code psect 138
colours 51
attributes, table of 52
settings 53
valid 51
values, table of 52
command files, XAC 92
command line driver 91
command lines
HLINK, long command lines 190
long 91, 196
verbose option 110
commands
block 65, 66
clipboard 69
HPDXA keyboard 65
comment block 68
commenting code 74

comments
block 68
C++ style 68, 74
compile menu 62, 78
compile and link 78
compile to .AS 78
compile to .OBJ 78
disable non-ANSI features 80
generate assembler listing 80
generate C source listing 80
identifier length 80
optimization 80
preprocess only to .PRE 79
stop on warnings 79
warning level 79
compiled stack 192
compiler
options 92
help 89
overview 19
release notes 89
technical support 89
compiler errors 102
format 101
compiler generated psects 137
compiling
to assembler file 78, 109
to executable file 78
to object file 78, 98
to preprocessor file 79
cond assembler control 177
conditional assembly 172
console I/O functions 158
table of 159
const
keyword 125
pointers 130
const psect 138
constants
C specifiers 118
continue button 62
copy 69
block 68
copyright notice 108, 116
COS 288
COSH 289
CPUTS 290
creating
libraries 195
projects 82
source files 71
CREF 161, 198
command line arguments 198
options 199
-Fprefix 199
-Hheading 199
-Llen 199
-Ooutfile 199
-Pwidth 200
-Sstoplist 200
-Xprefix 200
cromwell 200
available format options 200
cromwell application 30
cromwell options 201
-B 202
-C 201
-D 201
-E 202
-F 202
-Ikey 202
-L 202
-M 202
-Okey 202
-Pname 201
table of 201

-V 202
cross reference
disabling 178
enabling 179
generating 161, 198
list utility 198
cross reference listings 99
excluding header symbols 199
excluding symbols 200
headers 199
output name 199
page length 199
page width 200
CTIME 291
cut 69

D

Data book 168
data psect 138, 182
copying 182
data psects 197
data types 117
16 bit integer 121
32 bit IEEE floating point 122
32 bit integer 122
64 bit fast double 122
64 bit IEEE floating point 122
8 bit integer 121
bit 139, 170
char 121
floating point 122
int 121
long 122
short 121
db directive 172
dd directive 172
debug information 27, 78, 103, 112, 162
debugger
Lucifer 85, 205
setup 85
default libraries 92
defined symbols, linker 158
delete
block 68
selection 70
delta psect flag 185
dependency information 29, 78
design issues, hardware 111
df directive 172
DI 292
di() macro 141
directives
asm, C 149
assembler 168
table of 169
psect flags, table of 170
disabling interrupts 141
DIV 293
DOS
command line limitations 50, 91
commands 84
defining commands 87
free memory 71
shell 84
double precision floating point 122
double type
size of 100
downloading with Lucifer 85
ds directive 172
DS register 126
DUMP 28
dw directive 172

E

edit menu 72
- C colour coding 74
- comment/uncomment 74
- copy 72
- cut 72
- delete selection 73
- go to line 73
- hide 73
- indent 74
- outdent 74
- paste 73
- replace 73
- search 73
- set tab size 74
- show clipboard 73

edit window 63
editor 63
- auto indent mode 64
- autosave 71
- begin block 66
- block commands 65
- block hide/display 66
- C mode 65
- clear clipboard 70
- clipboard 65, 69
- colours 51
  - attributes, table of 52
  - settings, table of 53
  - values, table of 52
- comment/uncomment block 68
- commenting/uncommenting code 74
- content region 64
- copy 69, 72
  - block 68
- cut 69, 72
- delete block 68
- delete selection 70, 73
- end block 66
- frame area 64
- go to block end 66
- go to block start 66
- go to line 73
- hide 70, 73
  - block 69
- indent
  - block 68
  - mode 64
- indenting 74
- insert mode 64
- keyboard commands 65
- keys
  - help 89
  - table of 66, 67
- move block 68
- new 71
- opening recent files 72
- outdent block 68
- outdenting 74
- overwrite mode 64
- paste 70, 73
- read block 68
- replace text 73
- save 71
- save as 71
- search 65, 73
- selecting text 69
- show clipboard 70, 73
- status line 64
- syntax highlighting 74
- tab size 74
- write block 68
- zoom command 65

EI 292
ei() macro 141

ellipsis symbol 133
else directive 172
elseif directive 172
enabling interrupts 141
end block 66
end directive 169
endif directive 172
endm directive 173
enhanced S-Record 96
enhanced symbol files 186
environment variable
    HTC_ERR_FORMAT 101
    HTC_WARN_FORMAT 101
    LUCXA_ARGS 205
    TEMP 50
equ directive 167, 171
equating symbols 171
error files 101
    creating 185
error messages
    formatting 101
    LIBR 197
errors
    auto-fix 63
    format 101
    redirecting 101
ES register 126
EVAL_POLY 294
EXP 295
expand assembler control 177
expressions
    assembler 166
    relocatable 166
extern keyword 147
extrn directive 168

F

FABS 296
far
    keyword 80, 109, 125, 126
    variable 95
far pointers 126
farbss psect 139
farnvram psect 139
fast double
    floating point 164
    precision floating point 123
file formats 19, 76
    American Automation hex 96
    assembler 83
    assembler listing 80, 96
    Avocet 78
    binary 96
    C source 83
    C source listings 80
    command 92, 196
    creating with cromwell 200
    cross reference 161, 198
        listings 99
    DOS executable 187
    enhanced Motorola S-Record 96
    enhanced symbol 186
    Intel hex 103
    Intel OMF-51 106
    library 83, 113, 194, 196
    link 190
    map 83, 191
    Motorola hex 105
    object 83, 98, 187, 196
    preprocessor 79, 106
    project 82, 87
        files 82
    prototype 106

specifying 106, 162
S-Record files 105
symbol 186
symbol files 112
symbolic debug 27
Tektronix hex 109
TOS executable 187
UBROF 109
file menu 71
autosave 71
clear pick list 71
new 71
open 71
pick list 72
quit 71
save 71
save as 71
fix button 63
fixing errors 63
fixup 30
flags
psect 169
float type, size of 100
floating point
64 bit 122
double precision 122
IEEE library 104
fast double 164
library 104
precision 123
IEEE 164
floating point data types 75, 122
32-bit format 100
64-bit format 100
double precision 75
fast double precision 75
mantissa 122
single precision 75
FLOOR 297
fnconf directive 193
fnroot directive 192
FREE 298
FREXP 299
function
return values 133
16-bit 134
32-bit 134
64-bit 134
8-bit 133
structures 134
size limitations 111
function prototypes 157, 176
ellipsis 133
function qualifier
banked 141
noregsave 133
function return values 133
functions
argument passing 133
getch 159
getche 159
interrupt 139
interrupt qualifier 139
interrupts 141
jump table 155
kbhit 159
main 115, 117
putch 159
return values 133
returning from 140
signatures 157, 176
written in assembler 147

G

GETCH 300
getch function 159

GETCHE 300
getche function 159
GETS 301
global directive 148, 168
global optimization 27, 110
global psect flag 170
global symbols 182
GMTIME 302
go to line 73
grepping files 86

H

hardware
    design issues 111
    initialization 117
header files 20
heap psect 139
help button 63
help for XAC 103
help menu 88
    C library reference 89
    editor keys 89
    HI-TECH Software 88
    HPDXA 89
    release notes 89
    technical support 89
    XAC compiler options 89
hex files 156
    multiple 185
hide
    block 69, 70
    button 63
HLINK
    modifying options 46
HLINK options 183
    -Aclass=low-high 40, 184
    -Cpsect=class 185
    -Dsymfile 185
    -Eerrfile 185
    -F 185
    -Gspec 186
    -H+symfile 186
    -Hsymfile 186
    -Jerrcount 186
    -K 187
    -L 187
    -LM 187
    -Mmapfile 187
    -N 187
    -Nc 187
    -Ns 187
    -Ooutfile 187
    -Pspec 37, 187
    -Qprocessor 189
    -Sclass=limit[,bound] 189
    -Usymbol 189
    -Vavmap 190
    -Wnum 190
    -X 190
    -Z 190
hot keys 54
    HPDXA, table of 55
    windows, table of 56
HPDXA 49
    <<>> menu 70
    colours 51
        attributes, table of 52
        settings, table of 53
        values, table of 52
    command line arguments 50
    editor 63
    hardware requirements 50
    help 88
    hot keys 54
        table of 55

initialization file 51, 57
licence details 70
loading project file 50
menu bar 51
menus 70
mouse driver 51
moving windows 56
projects 80
pull down menus 51
quitting 71
resizing windows 56
screen
mode 51
resolution 50
selecting windows 54
starting 49
tutorial 58
version number 57
window hot keys 56
windows 49, 50
hpdxa.ini 51, 57
HTC_ERR_FORMAT 101
HTC_WARN_FORMAT 101
huge keyword 131
huge memory model 75, 96

I

I/O
console I/O functions 158
serial 158
STDIO 158
identifiers
assembler 165
length 80
IEEE floating point 164
format 75
IEEE floating point format 75
if directive 172
Implementation-defined behaviour 111
division and modulus 137
shifts 137
include assembler control 178
indent 74
block 68
mode 64
indenting code 74
ini file 19, 51, 57
setting colours in 51
in-line assembly 148
insert mode 64
int data types 121
Intel hex 156
internal RAM 126
interrupt
keyword 80, 109, 139
priorities 140, 142
interrupt functions 139
returning from 140
interrupts 140, 141, 217
<intrpt.h> 141
banked 141
CHANGE_VECTOR 141, 142, 144
di() 141
ei() 141
enabling 141
handling in C 139
handling macros 141
macros/functions, table of 141
RAM_VECTOR 141, 142, 144
READ_RAM_VECTOR 141, 142, 144
RETI instruction 140
ROM_VECTOR 141, 142
set_vector 141
interrupts disabling 141
irp directive 175

irpc directive 175
ISALNUM 304
ISALPHA 304
ISDIGIT 304
ISLOWER 304

J

Japanese character handling 150
JIS character handling 150
jis pragma directive 150
jump optimization 163
jump tables 155

K

KBHIT 306
kbhit function 159
keyword
    auto 131
    banked 141
    bit 119
    code 125, 127, 130
    const 125
    disabling non-ANSI 109
    extern 147
    far 80, 109, 125, 126
    huge 131
    interrupt 80, 109, 139
    near 80, 109, 125, 126
    noregsave 133
    persistent 125, 126, 129
    volatile 125

L

label field 167
labels
    aligning on boundaries 171
    temporary 165
large memory model 75, 96
LDEXP 307
LDIV 308
length of identifiers 80
LIBR 194, 195
    command line arguments 195
    error messages 197
    listing format 196
    long command lines 196
    module order 197
librarian 194
    command files 196
    command line arguments 195, 196
    error messages 197
    listing format 196
    long command lines 196
    module order 197
libraries
    adding files to 195
    C reference 89
    creating 195
    default 92
    deleting files from 195
    for XA 96, 97, 98
    format of 194
    large model 96
    linking 189
    listing modules in 195
    medium model 98
    module order 197
    naming convention 113
    order of 83
    scanning additional 104
    small model 98
    standard 113
    used in executable 187

library
difference between object file 194
manager 194
on-line manual 89
prefixes 104
Library functions
ABS 272
ACOS 273
ASCTIME 274
ASIN 276
ASSERT 277
ATAN 278
ATAN2 279
ATOF 280
ATOI 281
ATOL 282
BSEARCH 283
CALLOC 285
CEIL 286
CGETS 287
CHANGE_VECTOR 331
COS 288
COSH 289
CPUTS 290
CTIME 291
DI 292
DIV 293
EI 292
EVAL_POLY 294
EXP 295
FABS 296
FLOOR 297
FREE 298
FREXP 299
GETCH 300
GETCHE 300
GETS 301
GMTIME 302
ISALNUM 304
ISALPHA 304
ISDIGIT 304
ISLOWER 304
KBHIT 306
LDEXP 307
LDIV 308
LOCALTIME 309
LOG 311
LOG10 311
LONGJMP 312
MALLOC 314
MEMCHR 315
MEMCMP 316
MEMCPY 318
MEMMOVE 319
MEMSET 320
MODF 321
PERSIST_CHECK 322
PERSIST_VALIDATE 322
POW 323
PRINTF 324
PUTCH 327
PUTS 328
QSORT 329
RAM_VECTOR 331
RAND 332
READ_RAM_VECTOR 331
REALLOC 333
ROM_VECTOR 334
SCANF 335
SET_VECTOR 339
SETJMP 337
SIN 341
SINH 289
SPRINTF 342
SQRT 343
SRAND 344

SSCANF 345
STRCAT 346
STRCHR 347
STRCMP 348
STRCPY 349
STRCSPN 350
STRDUP 351
STRICHR 347
STRICMP 348
STRISTR 359
STRLEN 352
STRNCAT 353
STRNCMP 354
STRNCPY 355
STRNICMP 354
STRPBRK 356
STRRCHR 357
STRRICHR 357
STRSPN 358
STRSTR 359
STRTOK 360
TAN 361
TANH 289
TIME 362
TOASCII 363
TOLOWER 363
TOUPPER 363
VA_ARG 364
VA_END 364
VA_START 364
VPRINTF 324
VSCANF 335
VSPRINTF 342
VSSCANF 345
XTOI 366
licence
agreement 88
details 70
line numbers
C source 80
from assembler code 163
link addresses 34, 156, 182, 187
linker 29, 31, 181
command files 190
command line arguments 190
invoking 190
long command lines 190
modifying options 46
options from XAC 105
passes 194
symbols handled 182
linker defined symbols 158
linker errors
aborting 186
undefined symbols 187
linker options 36, 84, 183
-Aclass=low-high 40, 184, 188
-Cpsect=class 185
-Dsymfile 185
-Eerrfile 185
-F 185
-Gspec 186
-H+symfile 186
-Hsymfile 186
-I 187
-Jerrcount 186
-K 187
-L 187
-LM 187
-Mmapfile 187
-N 187
-Nc 187
-Ns 187
numbers in 184
-Ooutfile 187
-P 37

-Pspec 187
-Qprocessor 189
-Sclass=limit[, bound] 189
-Usymbol 189
-Vavmap 190
-Wnum 190
-X 190
-Z 190
linking programs 82
list files
assembler 28, 96
C source 23
generating 162
list, assembler control 178
listing file, C source code 99
little endian format 117
load addresses 34, 182, 187
local directive 165, 174
local psect flag 170
local psects 182
local symbols 110
suppressing 163, 190
local variables 131
auto 131
static 132
LOCALTIME 309
location counter 165
LOG 311
LOG10 311
LONGJMP 312
lucifer source level debugger (LUCXA) 205
LUCXA
@ command variants 214
command set 207
defaults 205
displaying arrays 213
expression forms 207
LUCXA_ARGS environment variable 205
memory mapping 216
LUCXA command 214
- 215
! 215
. 213
/ 215
= 215
@ 213
B 207
C 208
D 209
E 209
G 209
I 209
L 210
M 210
Q 210
R 211
S 211
T 212
U 212
W 212
X 212
LUCXA debugger 205
LUCXA options
-B 206
-P 205
-S 205
unix 205

M

macro directive 167, 173
macro names for DOS commands 87
macros
! character 176
% character 177
< and >characters 176

@ symbol 173
CHANGE_VECTOR 144
concatenation of arguments 173
defining 100
disabling in listing 178
expanding in listings 162, 177
interrupts 141
invoking 176
nul operator 174
predefined 150
preprocessor 100
repeat with argument 175
suppressing comments 174
undefining 109
unnamed 174
main function 115, 117
make 81
make menu 80
CPP include paths 84
CPP pre-defined symbols 84
library file list 83
linker options 84
load project 82
make 81
map file name 83
new project 82
object file list 83
objtohex options 84
output file name 82
re-link 82
re-make 82
rename project 82
save project 82
source file list 83
symbol file name 83
MALLOC 314
mantissa 122
map file options 78
map files 107, 187
call graphs 192
example of 191
generating 105
naming 83
processor selection 189
segments 191
sorting symbols 78
symbol tables in 187
width of 190
medium memory model 74, 98
MEMCHR 315
MEMCMP 316
MEMCPY 318
memmap 202
memmap options
-P 203
table of 203
-Wwid 203
MEMMOVE 319
memory
configuration 61
DOS 71
model 74, 114
huge 75, 96
huge library 104
large 75, 96
large library 104
medium 74, 98
medium library 104
small 74, 98
small library 104
specifying ranges 184
unused 187
usage 86, 107
memory mapping in Lucifer 216
MEMSET 320

menu
<<>> 57
compile 62, 78
edit 72
file 71
help 88
hot keys 396
make 80
mouse operation 53
options 74
run 84
setup 51, 57, 70
system 70
utility 86
menu bar 51
menus
accessing with keyboard 52
hot keys, for 54
HPDXA 70
pull down 51
mnemonics 163
MODF 321
modules
in library 194
list format 196
order in library 197
size limitations 111
used in executable 187
mouse
auto-repeat rate 70
ballistic threshold 70
driver 51, 71
sensitivity 57
move block 68
moving windows 56
multiple hex files 185
multiple source files 83

N

near keyword 80, 109, 125, 126
new files 71
next button 65
nocond assembler control 178
nogen assembler control 178
nojis pragma directive 150
nolist assembler control 178
non-volatile RAM 95
NOP insertion 162, 163
noregsave keyword 133
noxref assembler control 178
numbers
assembler 164
in C source 118
in linker options 184
numeric constants 163
nvram psect 139

O

object code, version number 187
object files 27, 83, 98
absolute 27, 187
displaying 28
including line numbers 163
precompiled 83
relocatable 27, 181
specifying object filenames 162
suppressing local symbols 163
symbol only 185
OBJTOHEX 30, 197
command line arguments 197
table of options 198
objtohex options 84
optimization 80
assembler 27, 163

code 158
explanation of 89
global 110
peephole 27, 105
options menu 74
float formats in printf 77
floating point type 75
long formats in printf 77
map and symbol file options 78
output file type 76
ROM and RAM addresses 76
save dependency information 78
sort map by address 78
source level debug info 78
suppress local symbols 78
org directive 171
outdent block 68, 74
outdenting code 74
output file formats 30, 187
HPDXA 76
specifying 106, 197
table of 30, 113
overlaid memory areas 187
overlaid psects 170
overwrite mode 64
ovrld psect flag 170

P

p1 application 24
pack pragma directive 150
pad bytes 162
page
length 162
width 163
page 0 memory model 114
page assembler control 177
parameter passing 133, 147
parameters 207
parser 24
output 24
paste 70
peephole optimization 27
peephole optimizer 105
PERSIST_CHECK 322
PERSIST_VALIDATE 322
persistent
keyword 125, 126, 129
type qualifier 95
variables 139
pointer types 128
table of sizes 128
pointers
to code 130
to const 130
POW 323
powerup routine 92, 117
pragma directives 150
jis 150
nojis 150
pack 150
printf_check 152
psect 152
strings 155
switch 155
table of 154
pre-compiled object files 83
predefined SFR names 165
predefined symbols
preprocessor 150
preprocessing 23, 79, 80, 106
assembler files 106
preprocessor
macros 100
output 23
path 84, 103

preprocessor directives 150
#asm 149
#endasm 149
table of 151
preprocessor symbols 84
predefined 150
previous button 65
PRINTF 324
printf
float support 77, 104
format checking 152
long support 77, 104
standard support 104
printf_check pragma directive 152
printing
floats 77
longs 77
processor selection 189
program sections 168
project 82
project files 82, 87
projects 80
building 81
creating 82
defining preprocessor symbols 84
libraries contained in 83
linker options 84
loading 82
map file name 83
object files contained in 83
options in 74
output file name 82
path to include files 84
re-building 82
renaming 82
saving 82
source files contained in 83
symbol file name 83
psect
bss 139, 182
code 138
const 138
data 138, 182, 197
farbss 139
farnvram 139
heap 139
nvram 139
rbit 139
rbss 139
rdata 138
stack 114, 139
strings 138
text 138
vectors 138
psect directive 168, 169
psect flags 169, 189
psect pragma directive 46, 152
psects 27, 31, 168, 181
absolute 170
alignment 171
basic kinds 181
class 40, 184, 185, 189
compiler generated 137
delta value of 185
grouping 35
linking 35, 181
local 182
maximum size of 171
overlaid 35
positioning 35
relocation 30
renaming 152
specifying address ranges 40, 188
specifying addresses 37, 184, 187
struct 134
types of 33

user defined 42, 152
pseudo-ops 168
table of 169
PSW register 140
table of values 143
public directive 168
pull down menus 51
pure psect flag 170
PUTCH 327
putch function 159
PUTS 328

Q

QSORT 329
qualifier
code 127
persistent 126
qualifiers
and auto variables 131
auto 131
code 155
const 129
far 126
near 126
persistent 129
strings 155
volatile 129
quick mode
assembler 163
quiet mode 108
quitting HPDXA 71
quoting of macro arguments 174

R

radix specifiers
assembler 163
C source 118
RAM 156
address 61
internal 126
non-volatile 95
RAM_VECTOR 331
RAND 332
rbit psect 139
rbss psect 132, 139
clearing 115
rdata psect 138
read block 68
READ_RAM_VECTOR 331
REALLOC 333
recently opened files 72
redirecting errors 102
redirecting options to XAC 92
register
bank switching 141
DS 126
ES 126
names 165
PSW 140
SFR names 165
usage 135
release notes 89
RELOC 186, 187
reloc psect flag 171
relocatable
object files 181
relocation 30, 181
relocation information
preserving 187
renaming psects 152
replacing text 73
rept directive 174
reset
code executed after 117
resizing windows 56

restore assembler control 178
RET instruction 140
RETI instruction 140
return values 133
ROM 156
    address 61
    checksum 77
ROM ranges 108
ROM_VECTOR 334
run menu 84
    auto download 85
    debugger 85
    DOS command 84
    DOS shell 84
    download 85
    download setup 85
run-time
    module 92, 114, 116
        files 83
    startup
        customizing 116
        using new 116

S

S1 format 156
save assembler control 178
saving files 71
SCANF 335
scroll bar 57
search 73
    button 65
search path, header files 103
searching files 73, 86
segment selector 186
segments 45, 186, 191
selecting text 69
serial I/O 158
serial port 159
set directive 167, 171
SET_VECTOR 339
SETJMP 337
setting tab size 74
setup menu 51, 57
SFR names 165
    predefined 165
SFRs 120
shift operations
    result of 137
show clipboard 70, 73
signat directive 148, 157, 176
signature checking 157
signatures 176
signed char variables 110
SIN 341
SINH 289
size error message, suppressing 163
size limitations
    functions 111
    modules 111
size psect flag 171
small memory model 74, 98
sound, enabling 57
source files 83
source listings 80
source modules 23
space psect flag 171
special characters 164
special funcition registers 120
SPRINTF 342
sprintf
    float support 77, 104
    long support 77, 104
SQRT 343
SRAND 344
S-Record files 105

SSCANF 345
stack psect 114, 139
standard libraries 113
standard symbols 84
startup module 92, 114, 116
    clearing bss 182
    data copying 182
statements
    assembler 167
static variables 132
status line 64
    indent/C mode indicator 64
    insert/overwrite indicator 64
    WordStar indicator 64
STDIO 158
STRCAT 346
STRCHR 347
STRCMP 348
STRCPY 349
STRCSPN 350
STRDUP 351
STRICHR 347
STRICMP 348
string search 73, 86
strings
    assembler 165
    qualifiers 155
strings pragma directive 155
strings psect 138
STRISTR 359
STRLEN 352
STRNCAT 353
STRNCMP 354
STRNCPY 355
STRNICMP 354
STRPBRK 356
STRRCHR 357
STRRICHR 357
STRSPN 358
STRSTR 359
STRTOK 360
struct psect 134
structure padding 150
structures 123
    bit fields 123
    qualifiers 124
switch
    code generation 155
    direct 155
    simple 155
switch pragma directive 155
symbol files 103, 112
    debug info 78
    enhanced 186
    generating 186
    local symbols in 190
    naming 83
    old style 185
    options 78
    removing local symbols from 78, 110
    removing symbols from 189
    source level 103
symbol tables 96, 103, 112, 187, 189
    assembler level 103
    Avocet format 96, 113
    for AVSIM 96, 103, 113
    for LUCXA 103
    sorting 187
    sorting addresses 78
    source level 103
symbols
    assembler 165
    ASXA generated 165
    equating 171
    global 182, 195
    linker defined 158

pre-defined 84
undefined 189
syntax highlighting 74
system menu 70

T

tab size 74
TAN 361
TANH 289
target code 216
inituart() 216
modifying 216
target.c 216
technical support 89
Tektronix hex files 109
temp path 20, 50
temporary labels 165
text psect 138
text search 73, 86
TIME 362
time 57
title assembler control 178
TOASCII 363
TOLOWER 363
TOUPPER 363
TSR programs 84
tutorial
compiling 62
errors 62
getting started 58
type modifiers
const 125
volatile 125
type qualifier
code 125, 127
far 125, 126
huge 131
near 125, 126
persistent 95, 125, 126
with auto variables 132
typographic conventions 17

U

UBROF files 109
uncomment block 68
uncommenting code 74
undefined symbols
assembler 163
unions 123
unsigned char variables 110
utilities 181
utility menu 86
ascii table 87
calculator 87
define user commands 87
memory usage map 86
string search 86

V

VA_ARG 364
VA_END 364
VA_START 364
variable argument list 133
variables
absolute 45, 132
accessing from assembler 148
auto 131
bit 119
char type 121
floating point types 122
int types 121
local 131
persistent 139

pointer types 128
static 132
vectors psect 138
verbose 110
version number 57
video card information 71
volatile keyword 125, 129
VPRINTF 324
VSCANF 335
VSPRINTF 342
VSSCANF 345

W

warning level 79, 110
setting 190
warnings 79
level displayed 110
suppressing 190
watchdog timer 117
window
edit 63, 64
error 62
windows
buttons in 57
moving 56
resize/move hot key 56
resizing 56
scroll bar in 57
selecting 54
zooming 57
word boundaries 171
WordStar
block commands 65
indicator 64
write block 68

X

XA
memory model 114
XA assembler language 163
functions 147
XAC
batch files 92
command files 92
command format 91
displaying help 103
file types 91
long command lines 91
options 92
predefined macros 150
redirecting options to 92
supported data types 117
XAC options
-A 92, 135, 156
-AAHEX 96
-ASMLIST 96
-AV 96, 103
-Bh 96
-BIN 96, 156
-Bl 96
-Bm 98
-Bs 98
-C 98, 156
-CLIST 99
-CR 99
-D 100
-DOUBLE 100
-E 100, 101
-G 103, 112, 205
-H 96, 103, 112, 205
-HELP 103
-I 103
-INTEL 103

-L 104, 105, 154
-LF 104
-M 105
-MOTOROLA 105
-N 105
-O 105, 106, 112, 156
-OMF51 106
-P 106
-PRE 106
-PROTO 106
-PSECTMAP 107, 156
-q 108
-RAM 108
-ROM 108
-S 109, 156, 157
-SA 109
-STRICT 109
-TEK 109
-U 109
-UBROF 109
-UNSIGNED 110, 121
-V 110
-W 110
-X 110
-Zg 110
XAC output formats
American Automation Hex 96, 112
Binary 112
binary 96
Bytecraft 112
Intel Hex 103, 106, 112
Intel OMF-51 106
Motorola Hex 105, 112
OMF-51 112
Tektronix Hex 109, 112
UBROF 109, 112
xref assembler control 179
XTOI 366

Z

zoom 57
zoom command 65

1 Introduction

2 Tutorials

3 Using HPDXA

4 XAC Command Line Compiler Driver

5 Features and Run-time Environment

6 The XA Macro Assembler

7 Linker and Utilities Reference Manual

8 Lucifer source level debugger

9 Error messages

10 Library functions

XAC Command Line Options

Option	Meaning
-A*spec*	Specify memory addresses for linking
-AAHEX	Generate an American Automation symbolic HEX file
-ASMLIST	Generate assembler .LST file for each compilation
-AV	Select AVOCET format symbol table
-BIN	Generate a Binary output file
-Bh	Select *huge* memory model
-Bl	Select *large* memory model
-Bm	Select *medium* memory model
-Bs	Select *small* medium model
-C	Compile to object files only
-CLIST	Generate C source listing file
-CR*file*	Generate cross-reference listing
-D*macro*	Define preprocessor macro on command line
-DOUBLE	Use 64 bit floating point for *double*
-E*[[+]file]*	Define format for compiler errors / redirect errors
-FDOUBLE	Use fast double float format
-G*file*	Generate source level symbol table
-H*file*	Generate enhanced symbol table
-HELP	Print summary of options
-INTEL	Generate an INTEL HEX format output file
-I*path*	Specify a directory pathname for include files
-L*library*	Specify a library to be scanned by the linker
-L-*option*	Specify -*option* to be passed directly to the linker
-M*file*	Request generation of a MAP file
-MOTOROLA	Generate a Motorola HEX format output file
-N*length*	Set identifier length to *length* (default is 31 characters)
-O	Enable post-pass optimization
-O*file*	Specify output filename and type
-OMF51	Produce an OMF-51 output file
-P	Pre-process assembler files
-PRE	Produce pre-processed source files
-PROTO	Generate function prototype information
-PSECTMAP	Display complete memory segment usage after linking
-q	Specify quiet mode
-ROM*ranges*	Specify ROM ranges for code
-RAM*ranges*	Specify RAM ranges for far data
-S	Compile to assembler source files only
-SA	Compile to Avocet AVMAC assembler source files
-STRICT	Enable strict ANSI keyword conformance
-TEK	Generate a Tektronix HEX format output file
-U*macro*	Undefine a predefined macro
-UBROF	Generate an UBROF format output file
-UNSIGNED	Make default character type *unsigned*
-V	Display compiler pass command lines
-W*level*	Set compiler warning level
-X	Eliminate local symbols from symbol table
-Zg	Enable global optimization in the code generator

HPDXA 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