

# HC(S)08 Compiler Manual

Revised: 5 November 2005



Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. CodeWarrior is a trademark or registered trademark of Freescale Semiconductor, Inc. in the United States and/or other countries. All other product or service names are the property of their respective owners.

Copyright © 2005 by Freescale Semiconductor, Inc. All rights reserved.

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals”, must be validated for each customer application by customer’s technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

## How to Contact Us

<b>Corporate Headquarters</b>	Freescale Semiconductor, Inc. 7700 West Parmer Lane Austin, TX 78729 U.S.A.
<b>World Wide Web</b>	<a href="http://www.freescale.com/codewarrior">http://www.freescale.com/codewarrior</a>
<b>Technical Support</b>	<a href="http://www.freescale.com/support">http://www.freescale.com/support</a>

# Table of Contents

---

## I Overview

## II Using the Compiler

<b>1</b>	<b>Introduction</b>	<b>27</b>
	Compiler environment	27
	Project directory	28
	Editor	28
	Creating a project for your application	28
	Using CodeWarrior to create a project	28
	Wizard	29
	CodeWarrior groups	39
	Analysis of some files in the project window	41
	Compilation with the Compiler	43
	Application Programs (Build Tools)	58
	Startup Command-Line Options	59
	Highlights	60
	CodeWarrior Integration of the Build Tools	60
	Combined or Separated Installations	60
	Target Settings preference panel	61
	Build Extras preference panel	65
	Assembler for HC08 preference panel	67
	Burner for HC08 preference panel	68
	Compiler for HC08 preference panel	69
	Importer for HC08 preference panel	71
	Linker for HC08 preference panel	72
	Libmaker for HC08 preference panel	73
	CodeWarrior Tips and Tricks	74
	Integration into Microsoft Visual Studio (Visual C++ V5.0 or later)	74

## Table of Contents

---

Integration as Additional Tools . . . . .	74
Integration with Visual Studio Toolbar . . . . .	76
Object-File Formats. . . . .	77
HIWARE Object-File Format . . . . .	77
ELF/DWARF Object-File Format . . . . .	77
Tools . . . . .	78
Mixing Object-File Formats . . . . .	78

## **2 Graphical User Interface 79**

Launching the Compiler . . . . .	79
Interactive Mode . . . . .	80
Batch Mode . . . . .	80
Tip of the Day . . . . .	81
Main Window . . . . .	82
Window Title . . . . .	82
Content Area . . . . .	82
Toolbar. . . . .	84
Status Bar . . . . .	85
Menu Bar. . . . .	85
File Menu. . . . .	86
Compiler Menu . . . . .	96
View Menu. . . . .	97
Help Menu . . . . .	97
Standard Types dialog box . . . . .	98
Option Settings dialog box . . . . .	99
Compiler Smart Control dialog box . . . . .	101
Message Settings dialog box . . . . .	103
Changing the Class associated with a Message. . . . .	105
Retrieving Information about an Error Message . . . . .	106
About... dialog box . . . . .	106
Specifying the Input File. . . . .	106
Use the Command Line in the Toolbar to Compile. . . . .	107
Message/Error Feedback . . . . .	107
Use Information from the Compiler Window . . . . .	108
Use a User-Defined Editor. . . . .	108

---

<b>3 Environment</b>	<b>109</b>
Current Directory . . . . .	110
Environment Macros . . . . .	111
Global Initialization File (mcutools.ini) . . . . .	112
Local Configuration File (usually project.ini) . . . . .	112
Paths . . . . .	113
Line Continuation . . . . .	114
Environment Variable Details . . . . .	115
COMPOPTIONS: Default Compiler Options . . . . .	116
COPYRIGHT: Copyright entry in object file . . . . .	117
DEFAULTDIR: Default Current Directory . . . . .	118
ENVIRONMENT: Environment File Specification . . . . .	119
ERRORFILE: Error filename Specification . . . . .	120
GENPATH: #include “File” Path . . . . .	122
INCLUDETIME: Creation Time in Object File . . . . .	123
LIBRARYPATH: ‘include <File>’ Path . . . . .	124
OBJPATH: Object File Path . . . . .	125
TEXTPATH: Text File Path . . . . .	126
TMP: Temporary Directory . . . . .	127
USELIBPATH: Using LIBPATH Environment Variable . . . . .	128
USERNAME: User Name in Object File . . . . .	129
<b>4 Files</b>	<b>131</b>
Input Files . . . . .	131
Source Files . . . . .	131
Include Files . . . . .	131
Output Files . . . . .	132
Object Files . . . . .	132
Error Listing . . . . .	132
Interactive Mode (Compiler window open) . . . . .	132
File Processing . . . . .	133
<b>5 Compiler Options</b>	<b>135</b>
Option Recommendation . . . . .	137

---

## Table of Contents

---

Compiler Option Details . . . . .	138
Option Groups . . . . .	138
Option Scopes . . . . .	139
Option Detail Description . . . . .	140
-!: filenames to DOS length . . . . .	143
-AddIncl: Additional Include File . . . . .	144
-Ansi: Strict ANSI . . . . .	146
-Asr: It is assumed that HLI code saves written registers . . . . .	147
-BfaB: Bitfield Byte Allocation . . . . .	149
-BfaGapLimitBits: Bitfield Gap Limit . . . . .	151
-BfaTSR: Bitfield Type Size Reduction . . . . .	153
-Cc: Allocate Constant Objects into ROM . . . . .	155
-Ccx: Cosmic Compatibility Mode for Space Modifiers and Interrupt Handlers . . . . .	157
-Ci: Tri- and Bigraph Support . . . . .	160
-Cni: No Integral Promotion . . . . .	163
-Cppc: C++ Comments in ANSI-C . . . . .	166
-Cq: Propagate const and volatile qualifiers for structs . . . . .	168
-Cs08: Generate Code for HCS08 . . . . .	170
-CswMaxLF: Maximum Load Factor for Switch Tables . . . . .	171
-CswMinLB: Minimum Number of Labels for Switch Tables . . . . .	173
-CswMinLF: Minimum Load Factor for Switch Tables . . . . .	175
-CswMinSLB: Minimum Number of Labels for Search Switch Tables . . . . .	177
-Cu: Loop Unrolling . . . . .	179
-Cx: No Code Generation . . . . .	182
-D: Macro Definition . . . . .	183
-Ec: Conversion from 'const T*' to 'T*' . . . . .	185
-Eencrypt: Encrypt Files . . . . .	187
-Ekey: Encryption Key . . . . .	189
-Env: Set Environment Variable . . . . .	190
-F (-Fh, -F1, -F1o, -F2, -F2o, -F6, or -F7): Object-File Format . . . . .	192
-Fd: Doubles are IEEE32. . . . .	194
-H: Short Help . . . . .	195
-I: Include File Path . . . . .	197

---

-La: Generate Assembler Include File . . . . .	199
-Lasm: Generate Listing File. . . . .	201
-Lasmc: Configure Listing File . . . . .	203
-Ldf: Log Predefined Defines to File . . . . .	205
-Li: List of Included Files . . . . .	207
-Lic: License Information. . . . .	209
-LicA: License Information about every Feature in Directory . . . . .	210
-LicBorrow: Borrow License Feature . . . . .	211
-LicWait: Wait until Floating License is Available from Floating License Server . . . . .	213
-Ll: Statistics about Each Function . . . . .	214
-Lm: List of Included Files in Make Format. . . . .	216
-LmCfg: Configuration of list of Included files in Make Format . . . . .	218
-Lo: Object File List . . . . .	221
-Lp: Preprocessor Output . . . . .	222
-LpCfg: Preprocessor Output configuration . . . . .	223
-LpX: Stop after Preprocessor. . . . .	225
-M (-Ms, -Mt): Memory Model. . . . .	226
-N: Display Notify Box . . . . .	227
-NoBeep: No Beep in Case of an Error. . . . .	229
-NoDebugInfo: Do not Generate Debug Information . . . . .	230
-NoEnv: Do not Use Environment . . . . .	232
-NoPath: Strip Path Info . . . . .	233
-O (-Os, -Ot): Main Optimization Target . . . . .	234
-Obfv: Optimize Bitfields and Volatile Bitfields. . . . .	236
-ObjN: Object filename Specification. . . . .	238
-Oc: Common Subexpression Elimination (CSE). . . . .	240
-OdocF: Dynamic Option Configuration for Functions . . . . .	242
-Of and-Onf: Create Sub-Functions with Common Code. . . . .	244
-Oi: Inlining. . . . .	247
-Oilib: Optimize Library Functions. . . . .	249
-Ol: Try to Keep Loop Induction Variables in Registers. . . . .	252
-Ona: Disable Alias Checking. . . . .	254
-OnB: Disable Branch Optimizer . . . . .	255
-Onbf: Disable Optimize Bitfields . . . . .	257

---

## Table of Contents

---

-Onbt: Disable ICG Level Branch Tail Merging . . . . .	259
-Onca: Disable any Constant Folding . . . . .	261
-Oncn: Disable Constant Folding in case of a New Constant . . . . .	263
-OnCopyDown: Do Generate Copy Down Information for Zero Values . . . . .	265
-OnCstVar: Disable CONST Variable by Constant Replacement . . . . .	267
-One: Disable any Low-level Common Subexpression Elimination . . . . .	268
-OnP: Disable Peephole Optimization. . . . .	270
-OnPMNC: Disable Code Generation for NULL Pointer to Member Check . . 272	
-Ont: Disable Tree Optimizer . . . . .	273
-OnX: Disable Frame Pointer Optimization . . . . .	280
-Or: Allocate Local Variables into Registers . . . . .	281
-Ou and -Onu: Optimize Dead Assignments . . . . .	283
-Pe: Preprocessing Escape Sequences in Strings . . . . .	285
-Pio: Include Files Only Once . . . . .	287
-Prod: Specify Project File at Startup . . . . .	289
-Qvtp: Qualifier for Virtual Table Pointers . . . . .	290
-Rp (-Rpe, -Rpt): Large Return Value Type . . . . .	291
-T: Flexible Type Management . . . . .	293
-V: Prints the Compiler Version. . . . .	299
-View: Application Standard Occurrence . . . . .	300
-WErrFile: Create "err.log" Error File . . . . .	302
-Wmsg8x3: Cut filenames in Microsoft Format to 8.3 . . . . .	304
-WmsgCE: RGB Color for Error Messages . . . . .	305
-WmsgCF: RGB Color for Fatal Messages . . . . .	306
-WmsgCI: RGB Color for Information Messages. . . . .	307
-WmsgCU: RGB Color for User Messages. . . . .	308
-WmsgCW: RGB Color for Warning Messages . . . . .	309
-WmsgFb (-WmsgFbi, -WmsgFbm): Set Message File Format for Batch Mode . . . . .	310
-WmsgFi (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode 312	
-WmsgFob: Message Format for Batch Mode . . . . .	314
-WmsgFoi: Message Format for Interactive Mode . . . . .	316



---

-WmsgFonf: Message Format for no File Information . . . . .	318
-WmsgFonp: Message Format for no Position Information . . . . .	320
-WmsgNe: Number of Error Messages . . . . .	322
-WmsgNi: Number of Information Messages . . . . .	323
-WmsgNu: Disable User Messages . . . . .	324
-WmsgNw: Number of Warning Messages . . . . .	326
-WmsgSd: Setting a Message to Disable . . . . .	327
-WmsgSe: Setting a Message to Error . . . . .	328
-WmsgSi: Setting a Message to Information . . . . .	329
-WmsgSw: Setting a Message to Warning . . . . .	330
-WOutFile: Create Error Listing File . . . . .	331
-Wpd: Error for Implicit Parameter Declaration . . . . .	333
-WStdout: Write to Standard Output . . . . .	335
-W1: No Information Messages . . . . .	337
-W2: No Information and Warning Messages . . . . .	338

## **6 Compiler Predefined Macros 339**

Compiler Vendor Defines . . . . .	339
Product Defines . . . . .	340
Data Allocation Defines . . . . .	340
Various Defines for compiler option settings . . . . .	341
Option Checking in C Code . . . . .	341
ANSI-C Standard Types 'size_t', 'wchar_t' and 'ptrdiff_t' Defines . . . . .	342
Division and Modulus . . . . .	345
Macro for HC08 . . . . .	345
Object-File Format Defines . . . . .	346
Bitfield Defines . . . . .	346
Bitfield Allocation . . . . .	346
Bitfield Type Reduction . . . . .	348
Sign of Plain Bitfields . . . . .	349
Macros for HC08 . . . . .	349
Type Information Defines . . . . .	350
Freescale HC08 Specific Defines . . . . .	352

---

<b>7</b>	<b>Compiler Pragmas</b>	<b>355</b>
	Pragma Details . . . . .	355
	#pragma CODE_SEG: Code Segment Definition . . . . .	357
	#pragma CONST_SEG: Constant Data Segment Definition . . . . .	360
	#pragma CREATE_ASM_LISTING: Create an Assembler Include File Listing . . . . .	363
	#pragma DATA_SEG: Data Segment Definition . . . . .	364
	#pragma INLINE: Inline Next Function Definition . . . . .	367
	#pragma INTO_ROM: Put Next Variable Definition into ROM. . . . .	368
	#pragma LINK_INFO: Pass Information to the Linker . . . . .	370
	#pragma LOOP_UNROLL: Force Loop Unrolling. . . . .	372
	#pragma mark: Entry in CodeWarrior IDE Function List . . . . .	373
	#pragma MESSAGE: Message Setting . . . . .	375
	#pragma NO_ENTRY: No Entry Code . . . . .	377
	#pragma NO_EXIT: No Exit Code . . . . .	379
	#pragma NO_FRAME: No Frame Code . . . . .	381
	#pragma NO_INLINE: Do not Inline Next Function Definition . . . . .	383
	#pragma NO_LOOP_UNROLL: Disable Loop Unrolling . . . . .	384
	#pragma NO_RETURN: No Return Instruction . . . . .	385
	#pragma NO_STRING_CONSTR: No String Concatenation during Preprocessing . . . . .	387
	#pragma ONCE: Include Once . . . . .	388
	#pragma OPTION: Additional Options. . . . .	389
	#pragma STRING_SEG: String Segment Definition . . . . .	392
	#pragma TEST_CODE: Check Generated Code. . . . .	394
	#pragma TRAP_PROC: Mark function as interrupt Function . . . . .	396
<b>8</b>	<b>ANSI-C Frontend</b>	<b>397</b>
	Implementation Features . . . . .	397
	Keywords . . . . .	397
	Preprocessor Directives . . . . .	397
	Language Extensions . . . . .	398
	The <code>__far</code> Keyword . . . . .	404
	<code>__near</code> Keyword. . . . .	409
	<code>__va_sizeof__</code> Keyword . . . . .	411

---

---

interrupt Keyword . . . . .	412
__asm Keyword . . . . .	412
Intrinsic Functions . . . . .	413
Implementation-Defined Behavior . . . . .	414
Translation Limitations . . . . .	415
ANSI-C Standard . . . . .	418
Integral Promotions . . . . .	418
Signed and Unsigned Integers . . . . .	418
Arithmetic Conversions . . . . .	419
Order of Operand Evaluation . . . . .	419
Rules for Standard-Type Sizes . . . . .	420
Floating-Type Formats . . . . .	420
Floating-Point Representation of 500.0 for IEEE . . . . .	421
Representation of 500.0 in IEEE32 Format . . . . .	422
Representation of 500.0 in IEEE64 Format . . . . .	423
Representation of 500.0 in DSP Format . . . . .	424
Volatile Objects and Absolute Variables . . . . .	425
Bitfields . . . . .	426
Signed Bitfields . . . . .	426
Segmentation . . . . .	427
Optimizations . . . . .	430
Peephole Optimizer . . . . .	431
Strength Reduction . . . . .	431
Shift Optimizations . . . . .	431
Branch Optimizations . . . . .	431
Dead-Code Elimination . . . . .	432
Constant-Variable Optimization . . . . .	432
Tree Rewriting . . . . .	432
Using Qualifiers for Pointers . . . . .	434
Defining C Macros Containing HLI Assembler Code . . . . .	436
Defining a Macro . . . . .	437
Using Macro Parameters . . . . .	439
Using the Immediate-Addressing Mode in HLI Assembler Macros . . . . .	439
Generating Unique Labels in HLI Assembler Macros . . . . .	440
Generating Assembler Include Files . . . . .	

## Table of Contents

---

(-La compiler option) . . . . .	441
<b>9 Generating Compact Code</b>	<b>451</b>
Compiler Options . . . . .	451
-Or: Register Optimization . . . . .	451
-Oi: Inlining: Inline Functions . . . . .	451
__SHORT_SEG Segments . . . . .	452
Defining I/O Registers . . . . .	453
Programming Guidelines . . . . .	454
Constant Function at a Specific Address . . . . .	454
HLI Assembly . . . . .	454
Post and Pre Operators in Complex Expressions . . . . .	455
Boolean Types . . . . .	456
printf() and scanf() . . . . .	457
Bitfields . . . . .	457
Struct Returns . . . . .	457
Local Variables . . . . .	458
Parameter Passing . . . . .	459
Unsigned Data Types . . . . .	459
Inlining and Macros . . . . .	459
Data Types . . . . .	460
Short Segments . . . . .	461
Qualifiers . . . . .	461
<b>10 HC(S)08 Backend</b>	<b>463</b>
Memory Models . . . . .	463
SMALL Model . . . . .	463
TINY Model . . . . .	463
Non-ANSI Keywords . . . . .	463
Data Types . . . . .	464
Scalar Types . . . . .	464
Floating-Point Types . . . . .	465
Bitfields . . . . .	466
Pointer Types and Function Pointers . . . . .	468
Structured Types and Alignment . . . . .	469

---

Object Size . . . . .	469
Register Usage . . . . .	469
Call Protocol and Calling Conventions . . . . .	469
HC08 Argument Passing . . . . .	470
HCS08 Argument Passing (used for the -Cs08 option) . . . . .	470
HC08 Return Values . . . . .	470
HCS08 Return Values (used for the -Cs08 Option) . . . . .	471
Returning Large Objects . . . . .	471
Stack Frames . . . . .	471
Pragma TRAP_PROC . . . . .	472
Interrupt Vector Table Allocation . . . . .	472
Segmentation . . . . .	473
Optimizations . . . . .	474
Volatile Objects . . . . .	476
Generating Compact Code with the CHC08 Compiler . . . . .	476
Compiler Options . . . . .	476
__SHORT_SEG Segments . . . . .	476
Defining I/O Registers . . . . .	478
<b>11 High-Level Inline Assembler for the Freescale HC(S)08</b>	<b>481</b>
Syntax . . . . .	481
C Macros . . . . .	482
Inline Assembly Language . . . . .	482
Register Indirect Addressing Mode . . . . .	483
Example . . . . .	483
Special Features . . . . .	484
Caller/Callee Saved Registers . . . . .	484
Reserved Words . . . . .	484
Pseudo-Opcodes . . . . .	485
Accessing Variables . . . . .	485
Address Notation . . . . .	485
H:X Instructions . . . . .	485
Constant Expressions . . . . .	486
Optimizing Inline Assembly . . . . .	486
Assertions . . . . .	486

## Table of Contents

---

Stack Adjust. . . . .	487
In & Gen Sets . . . . .	487

# III ANSI-C Library Reference

## 12 Library Files 493

Directory Structure . . . . .	493
How to Generate a Library . . . . .	493
Common Source Files . . . . .	493
Startup Files. . . . .	494
Startup Files for Freescale HC08. . . . .	495
Startup Files for Freescale HCS08 . . . . .	495
Library Files . . . . .	495

## 13 Special Features 497

Memory Management -- malloc(), free(), calloc(), realloc(); alloc.c, and heap.c. . . . .	497
Signals - signal.c . . . . .	497
Multi-byte Characters - mblen(), mbtowc(), wctomb(), mbstowcs(), wcstombs(); stdlib.c . . . . .	498
Program Termination - abort(), exit(), atexit(); stdlib.c. . . . .	498
I/O - printf.c. . . . .	498
Locales - locale.* . . . . .	500
ctype . . . . .	500
String Conversions - strtol(), strtoul(), strtod(), and stdlib.c . . . . .	500

## 14 Library Structure 501

Error Handling. . . . .	501
String Handling Functions . . . . .	501
Memory Block Functions . . . . .	502
Mathematical Functions . . . . .	502
Memory Management . . . . .	504
Searching and Sorting . . . . .	504

---

Character Functions . . . . .	505
System Functions . . . . .	505
Time Functions . . . . .	506
Locale Functions . . . . .	506
Conversion Functions . . . . .	507
printf() and scanf() . . . . .	507
File I/O . . . . .	507
<b>15 Types and Macros in the Standard Library</b>	<b>511</b>
errno.h . . . . .	511
float.h . . . . .	511
limits.h . . . . .	512
locale.h . . . . .	513
math.h . . . . .	515
setjmp.h . . . . .	515
signal.h . . . . .	515
stddef.h . . . . .	516
stdio.h . . . . .	517
stdlib.h . . . . .	518
time.h . . . . .	519
string.h . . . . .	519
assert.h . . . . .	519
stdarg.h . . . . .	520
ctype.h . . . . .	521
<b>16 The Standard Functions</b>	<b>523</b>
abort() . . . . .	524
abs() . . . . .	525
acos() and acosf() . . . . .	526
asctime() . . . . .	527
asin() and asinf() . . . . .	528
assert() . . . . .	529
atan() and atanf() . . . . .	530
atan2() and atan2f() . . . . .	531
atexit() . . . . .	532

---

## Table of Contents

---

atof() .....	533
atoi() .....	534
atol() .....	535
bsearch() .....	536
calloc() .....	538
ceil() and ceilf() .....	539
clearerr() .....	540
clock() .....	541
cos() and cosf() .....	542
cosh() and coshf() .....	543
ctime() .....	544
difftime() .....	545
div() .....	546
exit() .....	547
exp() and expf() .....	548
fabs() and fabsf() .....	549
fclose() .....	550
feof() .....	551
ferror() .....	552
fflush() .....	553
fgetc() .....	554
fgetpos() .....	555
fgets() .....	556
floor() and floorf() .....	557
fmod() and fmodf() .....	558
fopen() .....	559
fprintf() .....	561
fputc() .....	562
fputs() .....	563
fread() .....	564
free() .....	565
freopen() .....	566
frexp() and frexpf() .....	567
fscanf() .....	568
fseek() .....	569



## Table of Contents

---

fsetpos()	570
ftell()	571
fwrite()	572
getc()	573
getchar()	574
getenv()	575
gets()	576
gmtime()	577
isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), and isxdigit()	578
labs()	580
ldexp() and ldexpf()	581
ldiv()	582
localeconv()	583
localtime()	584
log() and logf()	585
log10() and log10f()	586
longjmp()	587
malloc()	588
mblen()	589
mbstowcs()	590
mbtowc()	591
memchr()	592
memcmp()	593
memcpy() and memmove()	594
memset()	595
mktime()	596
modf() and modff()	597
perror()	598
pow() and powf()	599
printf()	600
putc()	601
putchar()	602
puts()	603
qsort()	604

## Table of Contents

---

raise()	.606
rand()	.607
realloc()	.608
remove()	.609
rename()	.610
rewind()	.611
scanf()	.612
setbuf()	.613
setjmp()	.614
setlocale()	.615
setvbuf()	.616
signal()	.617
sin() and sinf()	.618
sinh() and sinhf()	.619
sprintf()	.620
sqrt() and sqrtf()	.624
srand()	.625
sscanf()	.626
strcat()	.630
strchr()	.631
strcmp()	.632
strcoll()	.633
strcpy()	.634
strcspn()	.635
strerror()	.636
strftime()	.637
strlen()	.639
strncat()	.640
strncmp()	.641
strncpy()	.642
strpbrk()	.643
strrchr()	.644
strspn()	.645
strstr()	.646
strtod()	.647

---

strtok()	648
strtol()	649
strtoul()	651
strxfrm()	652
system()	653
tan() and tanf()	654
tanh() and tanhf()	655
time()	656
tmpfile()	657
tmpnam()	658
tolower()	659
toupper()	660
ungetc()	661
va_arg(), va_end(), and va_start()	662
vfprintf(), vprintf(), and vsprintf()	663
wctomb()	664
wctombs()	665

## IV Appendices

<b>A</b>	<b>Porting Tips and FAQs</b>	<b>669</b>
	Migration Hints	669
	Porting from Cosmic	669
	Allocation of Bitfields	675
	Type Sizes and Sign of char	675
	@bool Qualifier	676
	@tiny and @far Qualifier for Variables	676
	Arrays with Unknown Size	676
	Missing Prototype	677
	_asm("sequence")	677
	Recursive Comments	677
	Interrupt Function, @interrupt	678
	Defining Interrupt Functions	678

## Table of Contents

---

Protecting Parameters in the OVERLAP Area . . . . .	681
How to Use Variables in EEPROM. . . . .	683
Linker Parameter File . . . . .	683
The Application . . . . .	684
General Optimization Hints . . . . .	686
Executing an Application from RAM . . . . .	687
ROM Library Startup File . . . . .	687
Generate an S-Record File. . . . .	688
Modify the Startup Code . . . . .	689
Application PRM File . . . . .	689
Copying Code from ROM to RAM . . . . .	690
Invoking the Application's Entry Point in the Startup Function . . . . .	690
Frequently Asked Questions (FAQs), Troubleshooting . . . . .	691
Making Applications . . . . .	691
Bug Reports. . . . .	698
Information . . . . .	698
Bug. . . . .	698
Critical Bug . . . . .	698
EBNF Notation . . . . .	698
Terminal Symbols . . . . .	699
Non-Terminal Symbols . . . . .	699
Vertical Bar . . . . .	699
Brackets . . . . .	699
Parentheses . . . . .	700
Production End . . . . .	700
EBNF Syntax. . . . .	700
Extensions . . . . .	700
Abbreviations, Lexical Conventions . . . . .	701
Number Formats. . . . .	701
Precedence and Associativity of Operators for ANSI-C. . . . .	702
List of all Escape Sequences . . . . .	704

## **B Global Configuration-File Entries 705**

[Options] Section. . . . .	705
DefaultDir . . . . .	705

---

[XXX_Compiler] Section . . . . .	706
SaveOnExit . . . . .	706
SaveAppearance . . . . .	706
SaveEditor . . . . .	706
SaveOptions . . . . .	707
RecentProject0, RecentProject1, ....	707
TipFilePos . . . . .	708
ShowTipOfDay . . . . .	708
TipTimeStamp . . . . .	708
[Editor] Section . . . . .	709
Editor_Name . . . . .	709
Editor_Exe . . . . .	709
Editor_Opts . . . . .	710
Example . . . . .	710

**C Local Configuration-File Entries 713**

[Editor] Section . . . . .	713
Editor_Name . . . . .	713
Editor_Exe . . . . .	714
Editor_Opts . . . . .	714
Example [Editor] Section . . . . .	714
[XXX_Compiler] Section . . . . .	715
RecentCommandLineX . . . . .	715
CurrentCommandLine . . . . .	715
StatusBarEnabled . . . . .	716
ToolbarEnabled . . . . .	716
WindowPos . . . . .	717
WindowFont . . . . .	717
Options . . . . .	718
EditorType . . . . .	718
EditorCommandLine . . . . .	719
EditorDDEClientName . . . . .	719
EditorDDETopicName . . . . .	720
EditorDDEServiceName . . . . .	720
Example . . . . .	720

## Table of Contents

---

**Index**

**723**

# Overview

---

The HC(S)08 Compiler manual describes the Compiler used for the Freescale 8-bit MCU (Microcontroller Unit) chip series. This document contains these major sections:

- Overview (this section): Description of the structure of this document and a bibliography of C language programming references
- Using the Compiler: Description of how to run the Compiler
- ANSI-C Library Reference: Description on how the Compiler uses the ANSI-C library
- Appendices: FAQs, Troubleshooting, and Technical Notes

Refer to the documentation listed below for details about programming languages.

- “American National Standard for Programming Languages – C”, ANSI/ISO 9899–1990 (see ANSI X3.159-1989, X3J11)
- “The C Programming Language”, second edition, Prentice-Hall 1988
- “C: A Reference Manual”, second edition, Prentice-Hall 1987, Harbison and Steele
- “C Traps and Pitfalls”, Andrew Koenig, AT&T Bell Laboratories, Addison-Wesley Publishing Company, Nov. 1988, ISBN 0-201-17928-8
- “Data Structures and C Programs”, Van Wyk, Addison-Wesley 1988
- “How to Write Portable Programs in C”, Horton, Prentice-Hall 1989
- “The UNIX Programming Environment”, Kernighan and Pike, Prentice-Hall 1984
- “The C Puzzle Book”, Feuer, Prentice-Hall 1982
- “C Programming Guidelines”, Thomas Plum, Plum Hall Inc., Second Edition for Standard C, 1989, ISBN 0-911537-07-4
- “DWARF Debugging Information Format”, UNIX International, Programming Languages SIG, Revision 1.1.0 (October 6, 1992), UNIX International, Waterview Corporate Center, 20 Waterview Boulevard, Parsippany, NJ 07054

- 
- “DWARF Debugging Information Format”, UNIX International, Programming Languages SIG, Revision 2.0.0 (July 27, 1993), UNIX International, Waterview Corporate Center, 20 Waterview Boulevard, Parsippany, NJ 07054
  - “System V Application Binary Interface”, UNIX System V, 1992, 1991 UNIX Systems Laboratories, ISBN 0-13-880410-9
  - 'Programming Microcontroller in C', Ted Van Sickle, ISBN 1878707140
  - 'C Programming for Embedded Systems', Kirk Zurell, ISBN 1929629044
  - 'Programming Embedded Systems in C and C ++', Michael Barr, ISBN 1565923545
  - 'Embedded C' Michael J. Pont ISBN 020179523X





# Using the Compiler

---

This section contains eleven chapters in the use and operation of the Compiler:

- Introduction: Description of the Compiler
- Graphical User Interface: Description of the Compiler's GUI
- Environment: Description of all the environment variables
- Files: Description of how the Compiler processes input and output files
- Compiler Options: Detailed description of the full set of Compiler options
- Compiler Predefined Macros: List of all macros predefined by the Compiler
- Compiler Pragmas: List of available pragmas
- ANSI-C Frontend: Description of the ANSI-C implementation
- Generating Compact Code: Programming advice for the developer to produce compact and efficient code.
- HC(S)08 Backend: Description of code generator and basic type implementation, also hints about hardware-oriented programming (optimizations, interrupt functions, etc.) specific for the Freescale HC(S)08.
- High-Level Inline Assembler for the Freescale HC(S)08: Description of the HLI Assembler for the HC08 and HCS08.



# Introduction

---

This chapter describes the Compiler as part of the CodeWarrior Development Studio used for the Freescale HC(S)08. The Compiler consists of a *Frontend*, which is language-dependent and a *Backend* that depends on the target processor, the HC(S)08. The Compiler is described in greater detail starting in Chapter 3. Chapters 1 and 2 describe the configuration and creation of projects using the Freescale HC(S)08.

The major sections of this chapter are:

- “Compiler environment” on page 27
- “Creating a project for your application” on page 28
- “Using CodeWarrior to create a project” on page 28
- “Compilation with the Compiler” on page 43
- “Application Programs (Build Tools)” on page 58
- “Startup Command-Line Options” on page 59
- “Highlights” on page 60
- “CodeWarrior Integration of the Build Tools” on page 60
- “Integration into Microsoft Visual Studio (Visual C++ V5.0 or later)” on page 74
- “Object-File Formats” on page 77

## Compiler environment

The Compiler can be used as a transparent, integral part of the Freescale CodeWarrior Development Studio. Using the CodeWarrior IDE is the recommended way to get your project up and running in minimal time. Alternatively, the Compiler can still be configured and used as a standalone application as a member of a suite of other Build Tool Utilities such as a Linker, Assembler, ROM Burner, Simulator or Debugger, etc.

In general, a Compiler translates source code such as from C source code files (\*.c) and header (\*.h) files into object-code (\*.o) files for further processing by a Linker. The \*.c files contain the programming code for the project’s application, and the \*.h files have data that is specifically targeted to a particular CPU chip or are interface files for functions. The Compiler can also directly generate an absolute (\*.abs) file that the Burner uses to produce an S-Record (\*.s19 or \*.sx) File for programming ROM memories.

It is also possible to mix assembly and C source code in a single project. See the High-Level Inline Assembler for the Freescale HC(S)08 chapter.

The typical configuration of the Compiler is its association with a Project directory and an Editor.

## Project directory

A project directory contains all of the environment files that you need to configure your development environment.

In the process of designing a project, you can either start from scratch by making your own project configuration (\*.ini) file and various layout files for your project for use with standalone project-building tools. On the other hand, you can let CodeWarrior coordinate and manage the entire project. Or, you can begin the construction of your project with CodeWarrior and also use the standalone build tools (Assembler, Compiler, Linker, Simulator/Debugger, etc.) that are included with the CodeWarrior suite.

---

**NOTE** The Build Tools are located in the `prog` folder in the CodeWarrior installation. The default location is `C:\Program Files\Freescale\CW for HC08\prog`.

---

## Editor

You can associate an editor, including the default text editor that is integrated into CodeWarrior, with the Compiler to enable Error Feedback. You can use the *Configuration* dialog box to configure the Compiler to select your choice of editors. Please refer to the Editor Settings dialog box section of this manual.

## Creating a project for your application

There are two primary ways to create a project with the CodeWarrior Development Studio:

- “Using CodeWarrior to create a project” on page 28

CodeWarrior can manage the entire project or just a part of it. The integrated Build Tools can also be used along with CodeWarrior.

- “Application Programs (Build Tools)” on page 58

You can use any or all of the standalone tools of the Build Tools that are included with the CodeWarrior installation. You can also employ CodeWarrior.

## Using CodeWarrior to create a project

CodeWarrior has a Wizard to easily configure and manage a project. You can get your project up and running by following a short series of steps to configure the project and to generate the basic files which are located in the project directory. You do not have to use the CodeWarrior IDE in order to use the standalone Build Tools (Compiler, Assembler, Linker, Burner, Simulator, etc.).

See “Application Programs (Build Tools)” on page 58 for information on using the standalone Compiler. Information on the other Build Tools can be found in user guides included with the CodeWarrior suite.

---

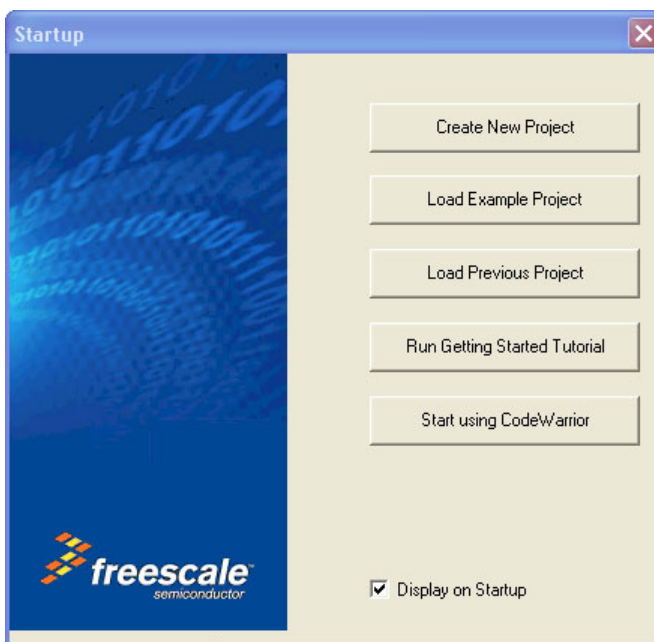
The following Wizard section will create and configure a basic CodeWarrior project that uses C source code. You can use the CodeWarrior interface to coordinate the integrated Build Tools.

## Wizard

This section creates a simple, bare-bones project using C source code. One of the means to create projects in CodeWarrior is the Wizard. Using the Wizard, a project only requires a few minutes to get up and running.

Start the HC08 CodeWarrior application. If CodeWarrior is opening from the Windows Start menu, the Startup dialog box appears inside the main window in CodeWarrior (Figure 1.1).

**Figure 1.1** Startup dialog box

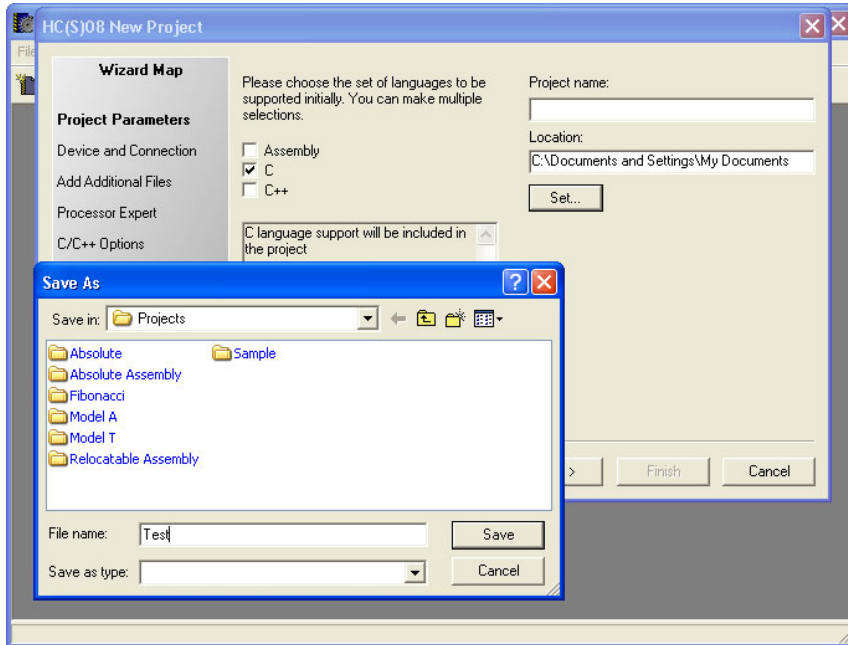


Press the *Create New Project* button. The *HC(S)08 New Project* dialog box appears showing the *Project Parameters* panel of the *Wizard Map*. If CodeWarrior is already running, select *New Project...* from the File menu (*File > New Project...*) instead. See Figure 1.2.

## Introduction

Using CodeWarrior to create a project

Figure 1.2 HC(S)08 New Project dialog box



Enter the *Project Parameters* for your project: Type the name for the project in the *Project name:* text box. In the event that you want a different location for the project directory than the default in the *Location:* text box, press *Set* and browse to the new location. There is no need to first prepare an empty folder, as CodeWarrior automatically creates its own folder - the *project directory*.

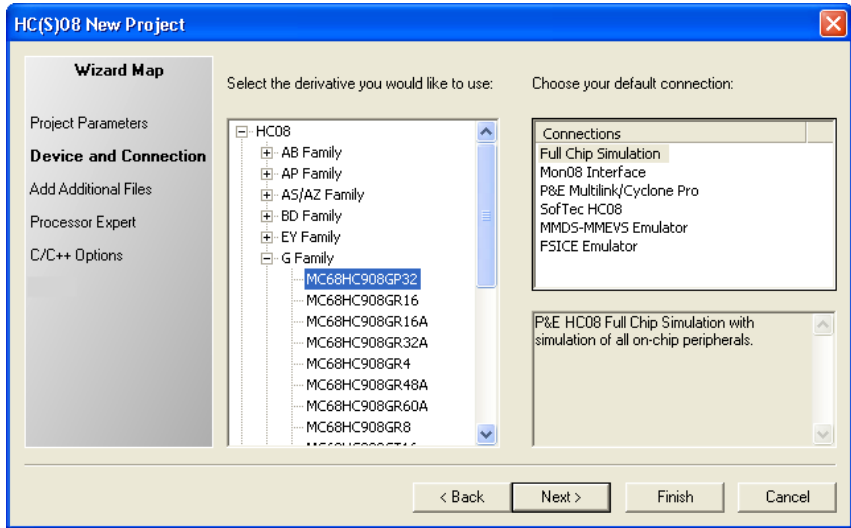
**NOTE** If you do not use the default *Location* for the project directory, you need not enter a name in the *Project name:* text box. Whatever you enter in the *File name:* text box will be entered into *Location* automatically.

CodeWarrior uses the default \*.mcp extension, so you do not have to explicitly append any extension to the filename.

For the programming language, check *C* and uncheck both *Assembly* and *C++*, in case that is not already set up that way. Press the *Save* and *Next >* buttons to close the dialog boxes.

The *Device and Connection* dialog box of the *Wizard Map* appears. (Figure 1.3).

Figure 1.3 Device and Connection dialog box

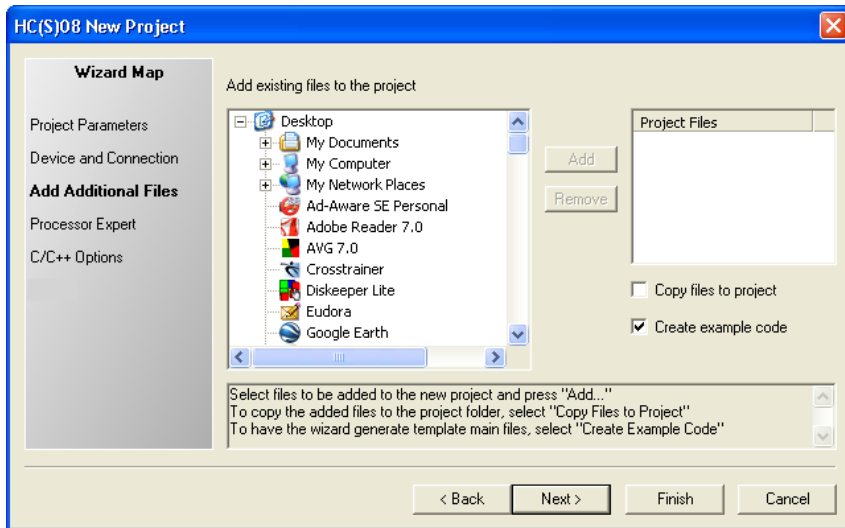


Select the desired CPU derivative for the project. Expand *HC08* and *G Family*. In this case, the *MC68HC908GP32* derivative is selected. For *Connections*, select the default - *Full Chip Simulation*. Press *Next >*. The *Add Additional Files* dialog box appears (Figure 1.4).

## Introduction

Using CodeWarrior to create a project

Figure 1.4 Add Additional Files dialog box

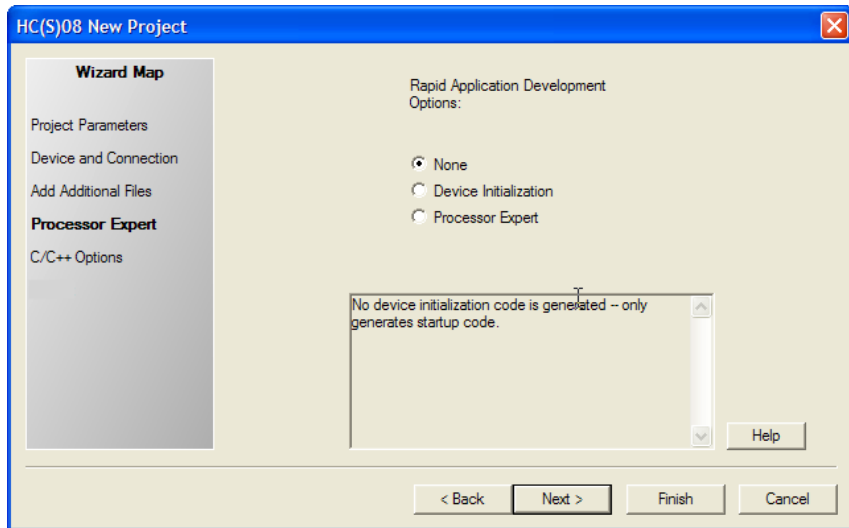


If you want to add any existing files to your project, you could browse the left pane - *Add existing files to the project* - for the files and press the *Add* button. The added files will appear in the right pane - *Project Files*. No user files are to be added for this project, so you can either uncheck the *Copy files to project* check box or make sure that no files are selected and leave this check box checked.

Check the *Create example code* check box. This enables template files including a *Sources* folder to be created in the project directory with some sample source-code files. Press *Next* >. The *Processor Expert* panel appears (Figure 1.5).



**Figure 1.5 Processor Expert panel**



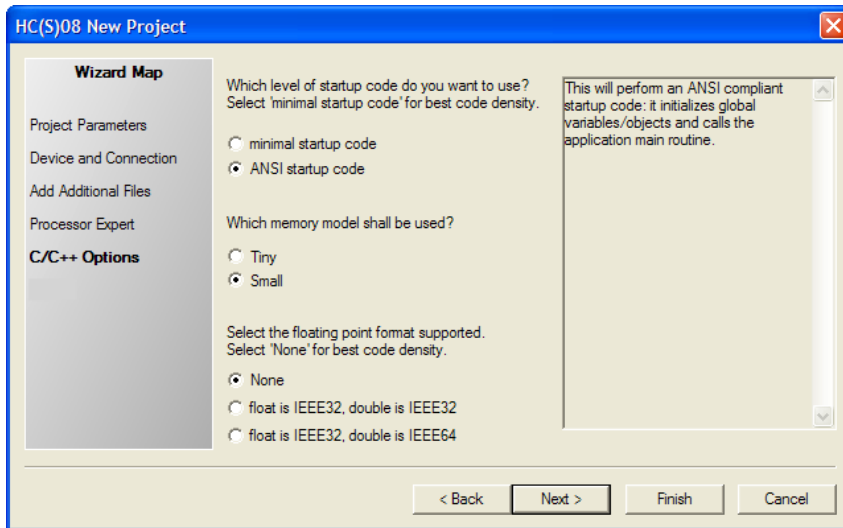
The default - *None* - is selected. For a simple demo project, you do not need the Rapid Application Development (RAD) tool - *Project Expert* - in the CodeWarrior Development Studio. We are now only interested in creating a bare-bones ANSI-C project. In practice, you would probably routinely use Processor Expert on account of its many advantages. Press *Next >*. The *C/C++ Options* panel appears (Figure 1.6).

## Introduction

Using CodeWarrior to create a project

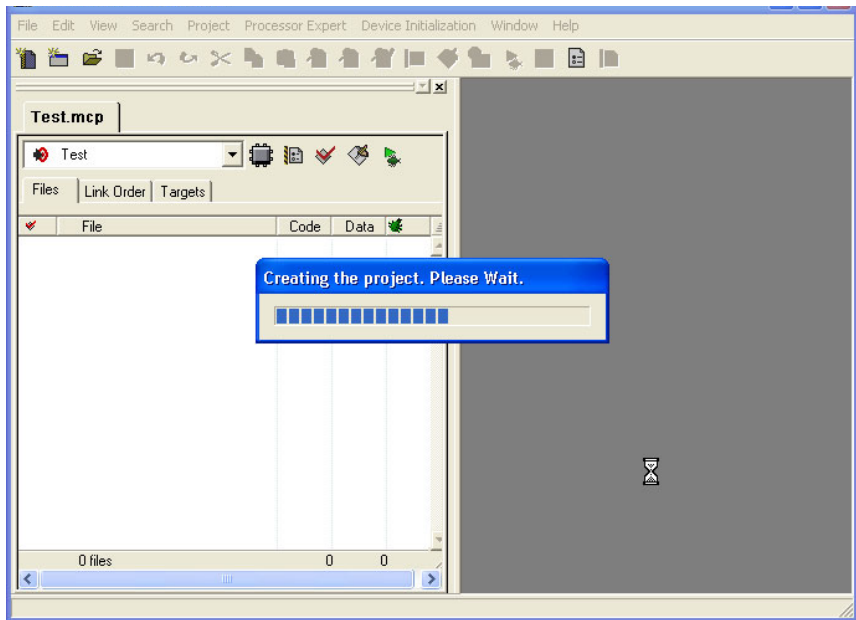
---

Figure 1.6 C/C++ Options panel



The three default selections for this panel are the usual entries for an ANSI-C project: *ANSI startup code*, *Small* memory model, and *None* for the floating-point numbers format. The default - *ANSI startup code* - is used. CodeWarrior automatically generates the startup and initialization routines for the specific derivative and calls the entry point into your ANSI-C project - the `main()` function. Use the integer-number format whenever possible in your projects, as floating-point numbers impose a severe speed-hit penalty. Press *Finish*. CodeWarrior now creates your project (Figure 1.7).

**Figure 1.7 Project creation**



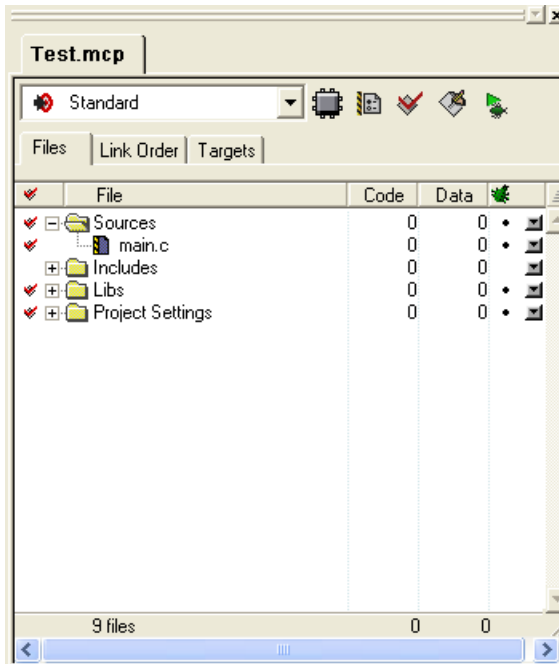
Using the Wizard, an HC(S)08 project is set up in about a minute or two. You can add additional components to your project afterwards. A number of files and folders are automatically generated in the root folder that was used in the project naming process. This folder is referred to in this manual as the project directory. The major GUI component for your project is the project window. The CodeWarrior project window appears (Figure 1.8).

## Introduction

*Using CodeWarrior to create a project*

---

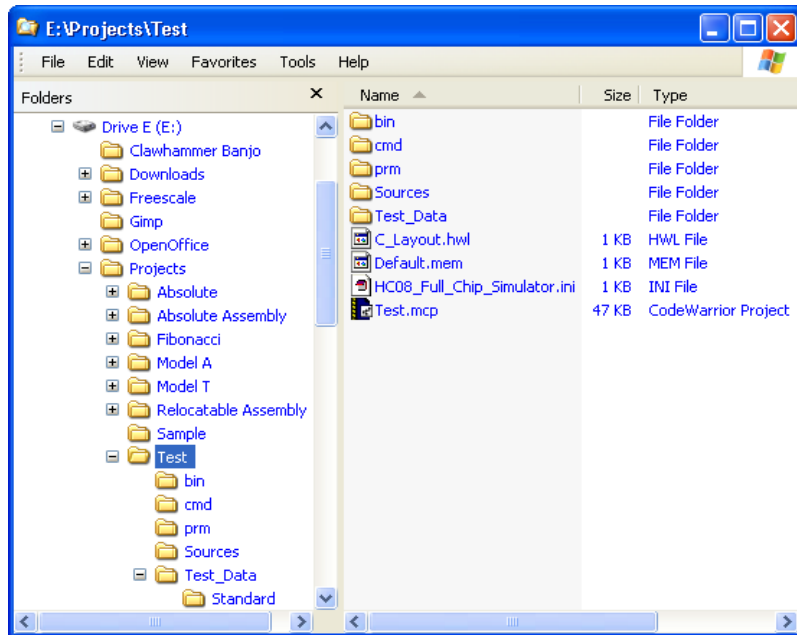
**Figure 1.8 CodeWarrior project window**



The project window contains four “folders,” but in reality they are not necessarily folders but abstract groups of files instead. If you were to examine the project directory that CodeWarrior generated after it created the project, you would see the actual folders and files that were generated for your project, as in Figure 1.9.

You should examine the folders and files that CodeWarrior created in the actual project directory so that you know they are in case you need to know. If you work with standalone tools such as a Compiler, Linker, Simulator/debugger, etc., you may need to specify the paths to these files. So it is best that you know their locations and functions. At this final stage of the Wizard, you could safely close the project and you can reopen it later. A CodeWarrior project reopens in the same configuration it was when it was last saved.

**Figure 1.9 Project directory in the Windows Explorer**



For this project, the name of the project directory and its path is:

E:\Projects\Test

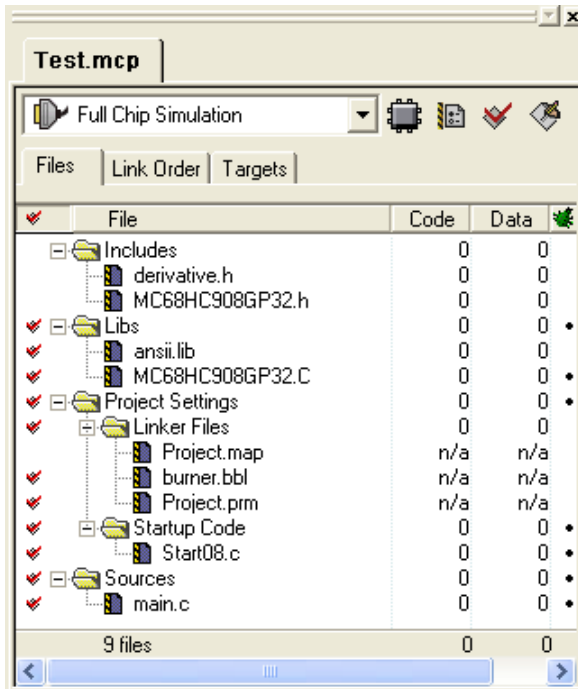
Inside the project directory is the master file for the project - Test .mcp. This is the file that you could use to open the project. Opening this master project file opens the CodeWarrior project in the same configuration it had when it was last saved.

Return to the CodeWarrior project window (Figure 1.10).

## Introduction

Using CodeWarrior to create a project

Figure 1.10 Project window showing the files that CodeWarrior created

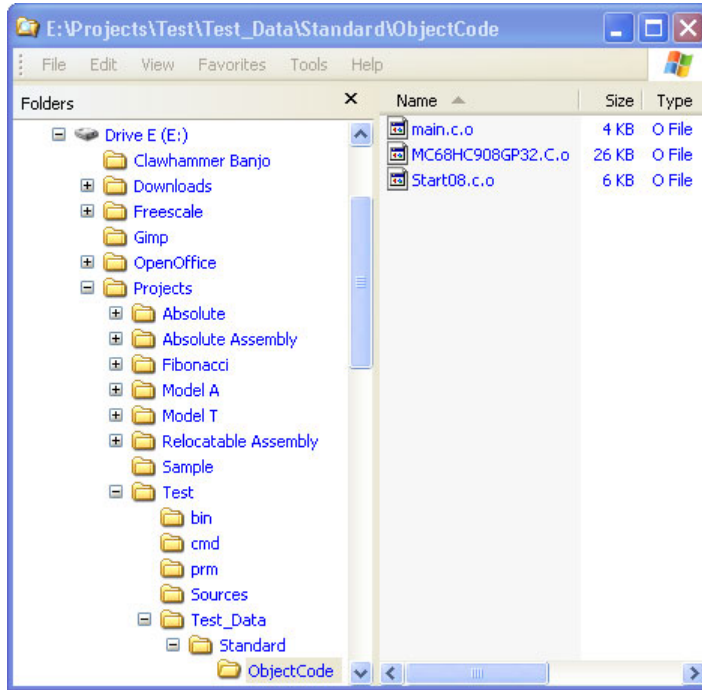


If you expand all the default groups, you would see two header files, a library file, three C-source files, and three others. Some of them have red check marks in the far-left column in the project window. These red marks will disappear when the project files are assembled, compiled, or linked under the control of the CodeWarrior IDE.

Using CodeWarrior you could compile the three C-source files separately, simultaneously, or in other combinations. You could compile them all simultaneously, but without linking their object code into an executable file. Select *Bring Up To Date* in the *Project* menu: *Project > Bring Up To Date (Ctrl+U)*. Notice that all of the red check marks have disappeared after the files were “brought up to date.”

An alternative means for processing C-source (\*.c) files (or \*.asm assembly or \*.lib library files) that have red check marks is to select one or more of these files in the project window and select *Project > Compile*. If you were to revisit the project directory with Windows Explorer after compilation, you would see that three object-code files were generated in the *ObjectCode* folder - one for each \*.c C-source file (Figure 1.11).

**Figure 1.11** Compilation generates object-code files



The compilation process for the C-source files is essentially complete at this point. However, compiled, binary object code is not very useful by itself. The object code needs to be further processed by a Linker in order to make it meaningful. This processing will be discussed later in this chapter ().

## CodeWarrior groups

There are four major groups in the project window for holding the project's files. It really does not matter much in which group a file resides as long as that file is somewhere in the project window. A file does not even have to be in any particular group. The groups do not have to correspond to any physical folders in the project directory. They are simply present in the project window for conveniently grouping files anyway you choose. You can add, rename, or delete files or groups, or you can move files or groups anywhere in the project window.

The default groups and their usual functions are:

- Includes

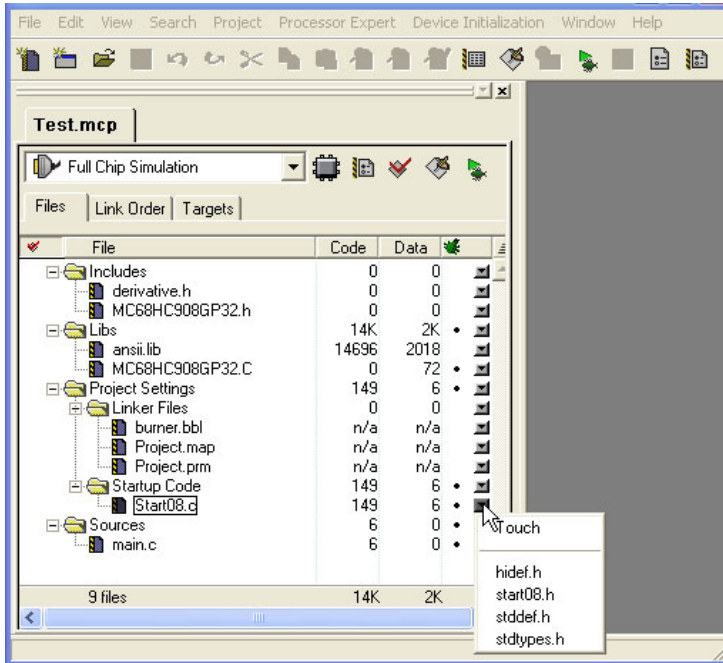
This group contains the some header files that CodeWarrior uses for its derivative-specific C source code files. The typical header files used by any C source file in the

## Introduction

### Using CodeWarrior to create a project

project window can be accessed by right- or left-clicking on the icon at the far right of any line in the project window that contains a \*.c file (Figure 1.12).

**Figure 1.12 Accessing header files for a \*.c file**



- Libs

The Libs group is where the `ansii.lib` library file and the C-source code file for the particular CPU derivative are located. `ansii.lib` holds the ANSI-C library of 'standardized' C functions. You can view the listing file for the functions and other objects contained in this library file at:

`<CodeWarrior_Installation>\lib\hc08c\lib\ansii.lst`. The `MC68HC908GP32.C` file is for the MC68HC908GP32 derivative.

- Project Settings

- Linker files

This group holds the burner file, the Linker mapping file, and the Linker PRM file.

- Startup code

This group holds the source code for the initialization and startup functions.



- Sources

This group contains the user's C-source code files.

## Analysis of some files in the project window

If you expand the “folders” - groups, actually, by clicking your mouse in the CodeWarrior project window, you can view all the default files that CodeWarrior generated. Those files marked by red check marks will remain checked until they are successfully assembled, compiled, or linked.

There are three C source code files visible in the project window:

- main.c,
- Start08.c, and
- MC68HC908GP32.C

Notice the bullets to the right of the C source files in the project window. If you ever experience the lack of viewing source code files in the *Source* pane in the simulator/debugger, always check first to see if these bullets are visible. If not, click and insert a bullet where they should be before debugging.

If the *Create example code* check box is checked in the Wizard while creating the project, the default CodeWarrior project has a sample C program in its *main.c* file in the *Sources* group which you will eventually replace with your own program. You could reuse the *main.c* file by reprogramming it or you could delete it entirely from the project and import your own C program files for your project.

As a precaution, you can see if the project is configured correctly and the source code is free of syntactical errors. It is not necessary that you do so, but you should make (build) the default project that CodeWarrior just created. We earlier just “brought up to date” the C-source files and the other files, so the red check marks are already gone at this point. Either select the *Make* button from the toolbar or from the *Project* menu, select (*Project > Make*). All of the red check marks will not be present and there should not be any error messages after a successful building of the project.

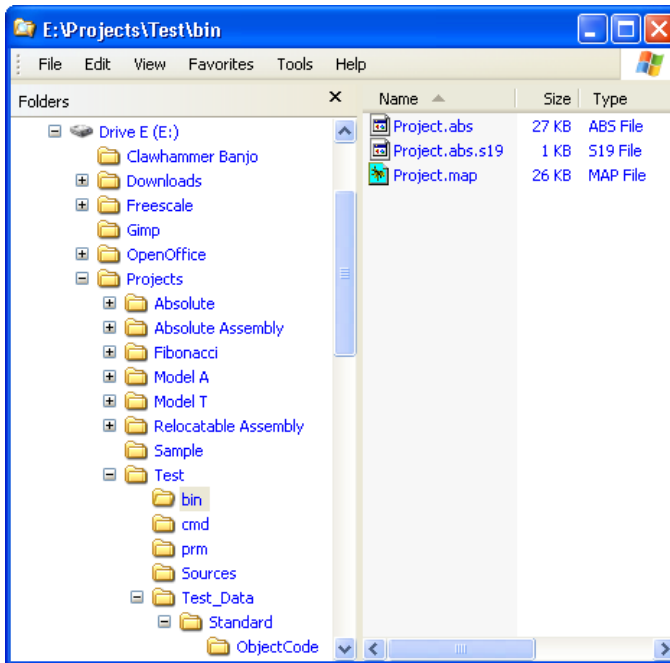
When the project is being built, the C source code is recompiled (if necessary), the object files are linked together, and the CPU derivative's ROM and RAM memory area are allocated by the Linker according to the settings in the PRM file. When everything goes OK in your project, there will be \*.abs and \*.abs.s19 files generated and placed in the bin subfolder of the project directory. The \*.abs.s19 file can be used to program ROM memories (Figure 1.13).

## Introduction

Using CodeWarrior to create a project

---

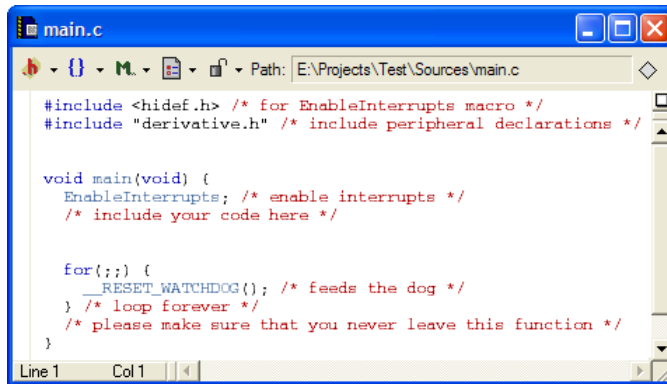
**Figure 1.13 bin subfolder of the project directory in Windows Explorer after a Make**



The \*.map file is the Linker Map file that indicates how the Linker allocated the memory areas for the project and contains other useful information. In addition, the object files (with the \*.o file extension) generated after compiling the C source files are contained in the ObjectCode subfolder in the <target\_name>\_Data folder.

Double click on the main.c file in the Sources group. The editor in CodeWarrior opens the default main.c file in the project window that CodeWarrior generated (Figure 1.14).

Figure 1.14 Default main.c file



```
#include <hidef.h> /* for EnableInterrupts macro */
#include "derivative.h" /* include peripheral declarations */

void main(void) {
    EnableInterrupts; /* enable interrupts */
    /* include your code here */

    for(;;) {
        __RESET_WATCHDOG(); /* feeds the dog */
    } /* loop forever */
    /* please make sure that you never leave this function */
}
```

You can use the editor integrated into CodeWarrior for writing your C (\*.c and \*.h) source files and add them to your project. Along the development phase, you can test your source code by building and possibly simulating/debugging your application.

The “Graphical User Interface” chapter provides information about configuring the options for and the feedback messages from the Compiler and other Build Tools.

## Compilation with the Compiler

It is also possible to use the HC(S)08 Compiler as a standalone compiler. This tutorial does not create an entire project with the Build Tools, but instead uses parts of a project already created by the CodeWarrior *Wizard*. CodeWarrior can create, configure, and manage a project much easier and quicker than using the Build Tools. However, the Build Tools could also create and configure a project from scratch. Instead, we will create a new project directory for this project, but will make use of some files already created in the previous project.

A Build Tool such as the Compiler makes use of a project directory file for configuring and locating its generated files. The folder that is properly configured for this purpose is referred to by a Build Tool as the “*current directory*.”

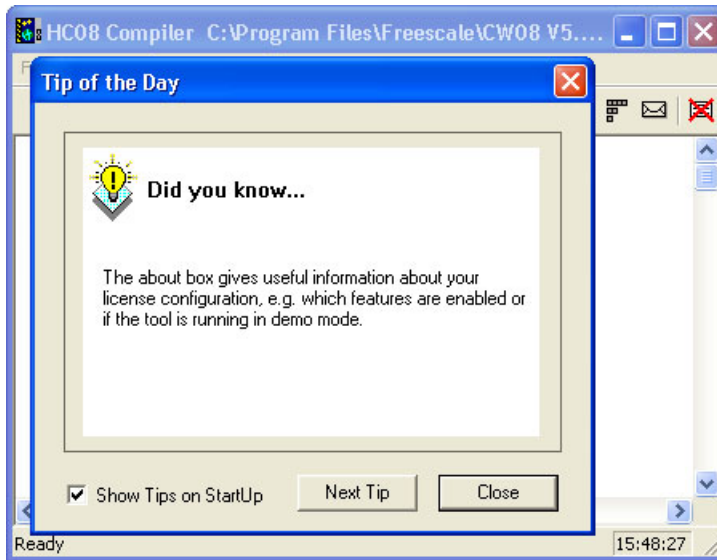
Start the Compiler. You can do this by opening the `chc08.exe` file in the `prog` folder in the HC08 CodeWarrior installation. The Compiler opens (Figure 1.15).

## Introduction

### Compilation with the Compiler

---

Figure 1.15 HC08 Compiler opens...



Read any of the Tips if you choose to and then press *Close* to close the *Tip of the Day* dialog box.

## Configuring the Compiler

A Build Tool, such as the Compiler, requires information from configuration files. There are two types of configuration data:

- Global

This data is common to all Build Tools and projects. There may be common data for each Build Tool (Assembler, Compiler, Linker, ...) such as listing the most recent projects, etc. All tools may store some global data into the `mcutools.ini` file.

The tool first searches for this file in the directory of the tool itself (path of the executable). If there is no `mcutools.ini` file in this directory, the tool looks for an `mcutools.ini` file located in the MS WINDOWS installation directory (e.g. `C:\WINDOWS`). See Listing 1.1.

### Listing 1.1 Typical locations for a global configuration file

---

```
\CW installation directory\prog\mcutools.ini - #1 priority  
C:WINDOWS\mcutools.ini - used if there is no mcutools.ini file above
```

---

If a tool is started in the C:\Program Files\CW08 V5\prog directory, the initialization file in the same directory as the tool is used.

C:\Program Files\CW08 V5\prog\mcutools.ini).

But if the tool is started outside the CodeWarrior installation directory, the initialization file in the Windows directory is used. For example, (C:\WINDOWS\mcutools.ini).

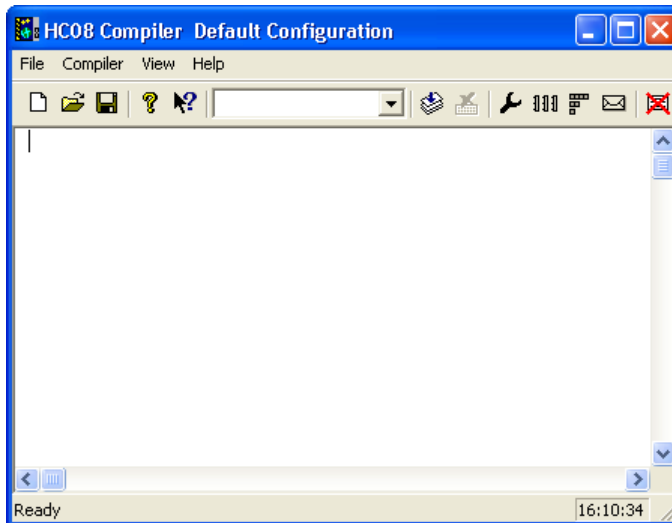
For information about entries for the global configuration file, see Global Configuration-File Entries in the Appendices.

- Local

This file could be used by any Build Tool for a particular project. For information about entries for the local configuration file, see Local Configuration-File Entries in the Appendices.

After opening the compiler, you would load the configuration file for your project if it already had one. However, you will create a new configuration file and save it so that when the project is reopened, its previously saved configuration state will be used. From the *File* menu, select *New / Default Configuration*. The *HC08 Compiler Default Configuration* dialog box appears (Figure 1.16)

**Figure 1.16 HC08 Compiler Default Configuration dialog box**



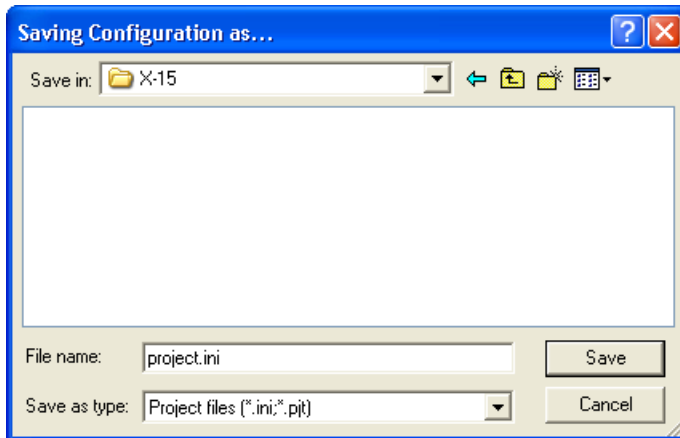
Now save this configuration in a newly created folder that will become the project directory. From the *File* menu, select *Save Configuration* (or *Save Configuration As...*). A *Saving Configuration as...* dialog box appears. Navigate to the folder of your choice and create and name a folder and filename for the configuration file (Figure 1.17).

## Introduction

### Compilation with the Compiler

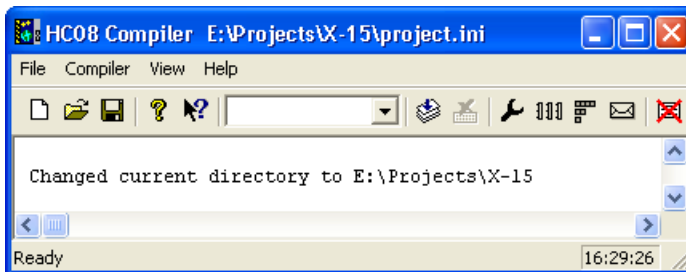
---

**Figure 1.17 Loading configuration dialog box**



Press *Open* and *Save*. The current directory of the HC08 Compiler changes to your new project directory (Figure 1.18).

**Figure 1.18 Compiler's current directory switches to your project directory...**



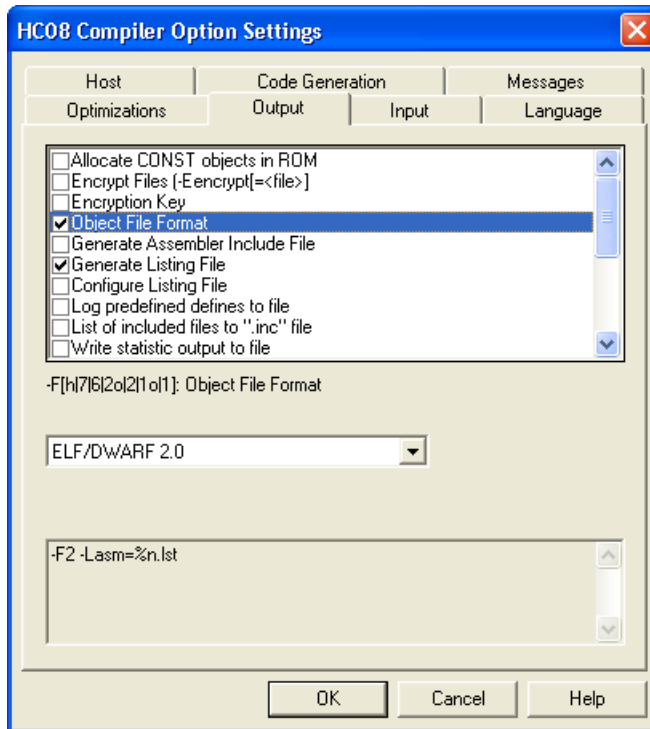
If you were to examine the project directory with the Windows Explorer at this point, it would only contain the *project.ini* configuration file that you just created. If you further examined the contents of the project's configuration file, you would notice that it now contains the *[CHC08 Compiler]* portion of the *project.ini* file in the *prog* folder where the Build Tools are located. Any options added to or deleted from your project by any Build Tool would be placed into or deleted from this configuration file in the appropriate section for each Build Tool.

If you want some additional options to be applied to all projects, you can take care of that later by changing the *project.ini* file in the *prog* folder.

You now set the object file format that you intend to use (HIWARE or ELF/DWARF). Select the menu entry *Compiler > Options... > Options*. The Compiler

displays the *HC08 Compiler Option Settings* dialog box. Select the *Output* tab (Figure 1.19).

**Figure 1.19** HC08 Compiler Option Settings dialog box



In the *Output* panel, select the check boxes labeled *Generate Listing File* and *Object File Format*. For the *Object File Format*, select the *ELF/DWARF 2.0* in the pull-down menu. Press *OK* to close the *HC08 Compiler Option Settings* dialog box.

Save the changes to the configuration by:

- selecting *File > Save Configuration (Ctrl + S)* or
- pressing the *Save* button on the toolbar.

## Input Files

Now that the project's configuration is initially set, you could compile an C source-code file. However, this project does not contain any source-code files at this point. You could create C source (\*.c) and header (\*.h) files from scratch for the project. However, for

## Introduction

### Compilation with the Compiler

---

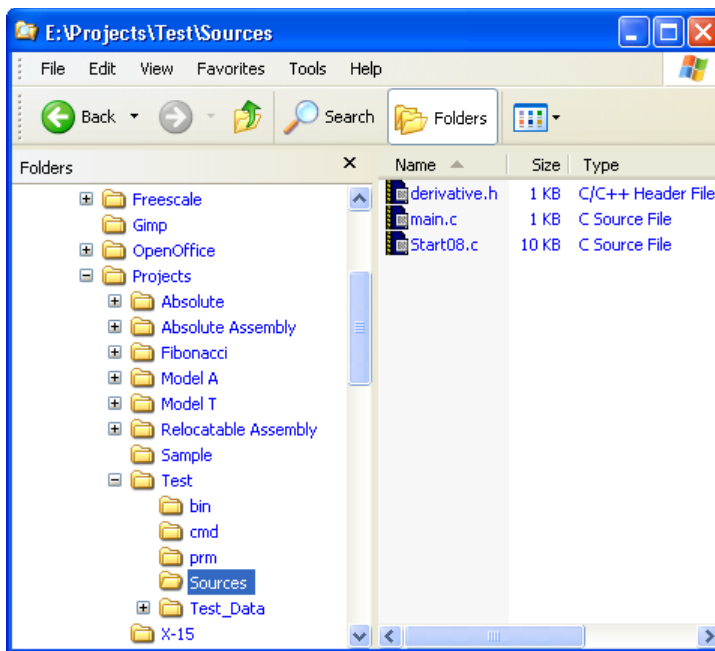
simplicity's sake, you can copy and paste the C source files from another project in order to avoid having to make your own.

The project created by the Wizard in the previous section used three C source files plus a number of header files. Two of the C-source files are located in the *Sources* folder from the *Test* project created in the previous section. The path for the remaining C-source file - MC68HC908GP32.C - is:

```
<CodeWarrior installation>\lib\HC08c\src
```

Copy-and-paste that *Sources* folder into the project directory for this project (X-15) . (See Figure 1.20.)

**Figure 1.20 Project files**



Now there are four files in the project:

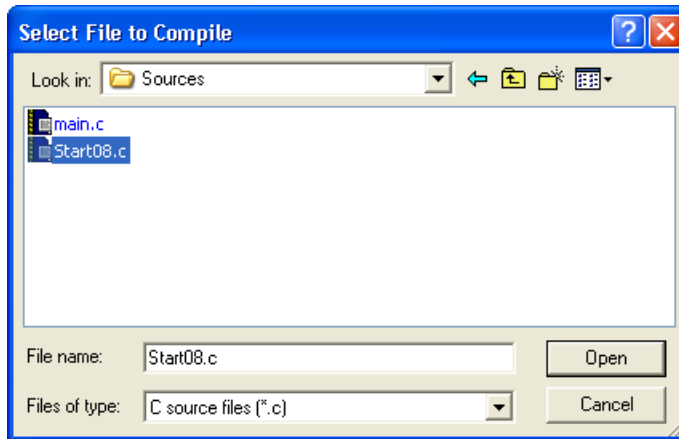
- the `project.ini` configuration file in the project directory and
- in the *Sources* folder:
  - `derivative.h`,
  - `main.c`, and  
The user's program
  - `Start08.c`.  
The startup and initialization routines



## Compiling the C source-code files

Let's compile one of the C-source files, say the `Start08.c` file. From the *File* menu, select *Compile*. The *Select File to Compile* dialog box appears (Figure 1.21).

Figure 1.21 Select File to Compile dialog box

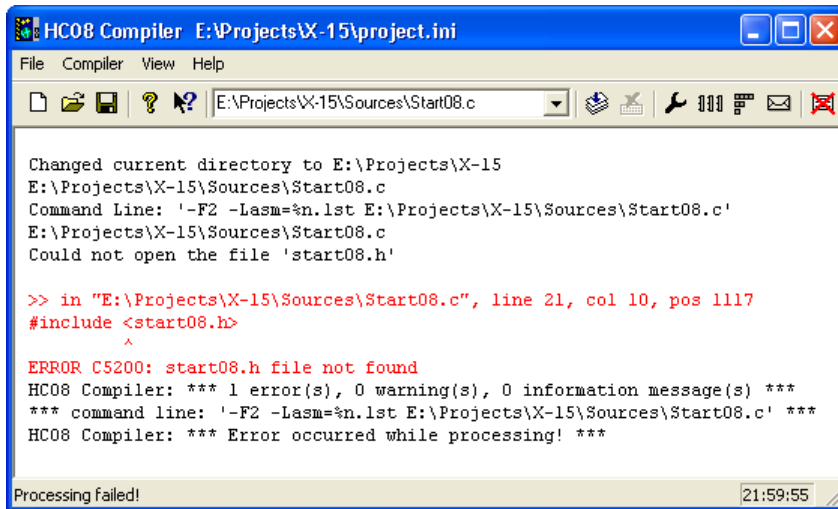


Browse to the *Sources* folder in the project directory and select the *Start08.c* file. Press *Open* and the *Start08.c* file should start compiling (Figure 1.22). Or maybe not....

## Introduction

### Compilation with the Compiler

Figure 1.22 Results of compiling the Start08.c file...



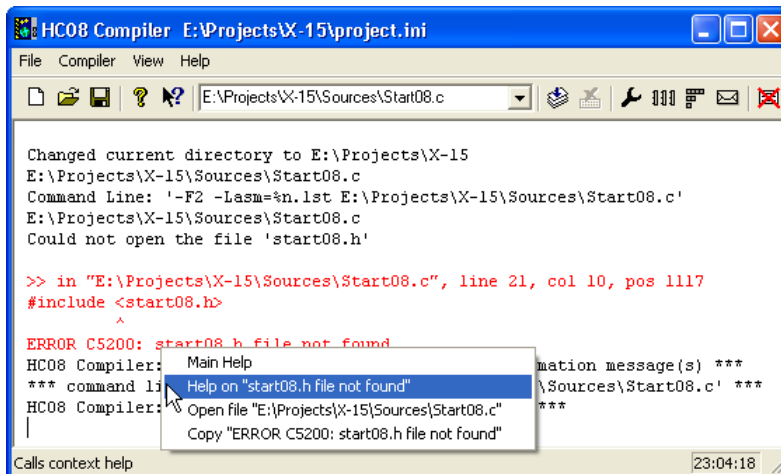
```
HC08 Compiler E:\Projects\X-15\project.ini
File Compiler View Help
E:\Projects\X-15\Sources\Start08.c
Changed current directory to E:\Projects\X-15
E:\Projects\X-15\Sources\Start08.c
Command Line: '-F2 -Lasm=%n.lst E:\Projects\X-15\Sources\Start08.c'
E:\Projects\X-15\Sources\Start08.c
Could not open the file 'start08.h'

>> in "E:\Projects\X-15\Sources\Start08.c", line 21, col 10, pos 1117
#include <start08.h>
      ^
ERROR C5200: start08.h file not found
HC08 Compiler: *** 1 error(s), 0 warning(s), 0 information message(s) ***
*** command line: '-F2 -Lasm=%n.lst E:\Projects\X-15\Sources\Start08.c' ***
HC08 Compiler: *** Error occurred while processing! ***

Processing failed! 21:59:55
```

The project window provides positive feedback information about the compilation process or generates error messages if the compiling was unsuccessful. In this case an error message was generated - the **C5200: 'FileName' file not found** message. If you right-click on the text about the error message, a context menu appears (Figure 1.23).

Figure 1.23 Context menu



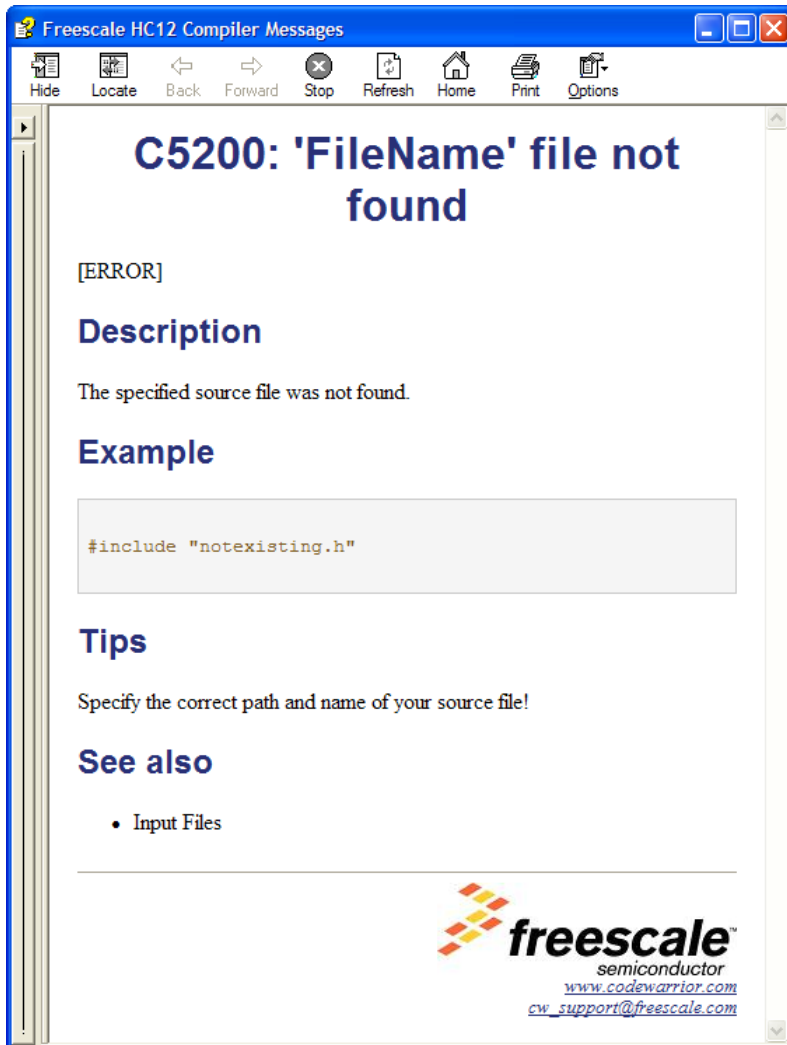
```
HC08 Compiler E:\Projects\X-15\project.ini
File Compiler View Help
E:\Projects\X-15\Sources\Start08.c
Changed current directory to E:\Projects\X-15
E:\Projects\X-15\Sources\Start08.c
Command Line: '-F2 -Lasm=%n.lst E:\Projects\X-15\Sources\Start08.c'
E:\Projects\X-15\Sources\Start08.c
Could not open the file 'start08.h'

>> in "E:\Projects\X-15\Sources\Start08.c", line 21, col 10, pos 1117
#include <start08.h>
      ^
ERROR C5200: start08.h file not found
HC08 Compiler: *** 1 error(s), 0 warning(s), 0 information message(s) ***
*** command line: '-F2 -Lasm=%n.lst E:\Projects\X-15\Sources\Start08.c' ***
HC08 Compiler: *** Error occurred while processing! ***

Calls context help 23:04:18
```

Select *Help on 'FileName' file not found* and help for the C5200 error message appears (Figure 1.24).

Figure 1.24 C5200 error message help



## Introduction

### Compilation with the Compiler

---

The *Tips* portion in the *Help for the C5200 error* states that you should specify the correct paths and names for the source files. The header file that the Compiler could not find is contained in the following folder:

```
<CodeWarrior installation folder>\lib\hc08\include
```

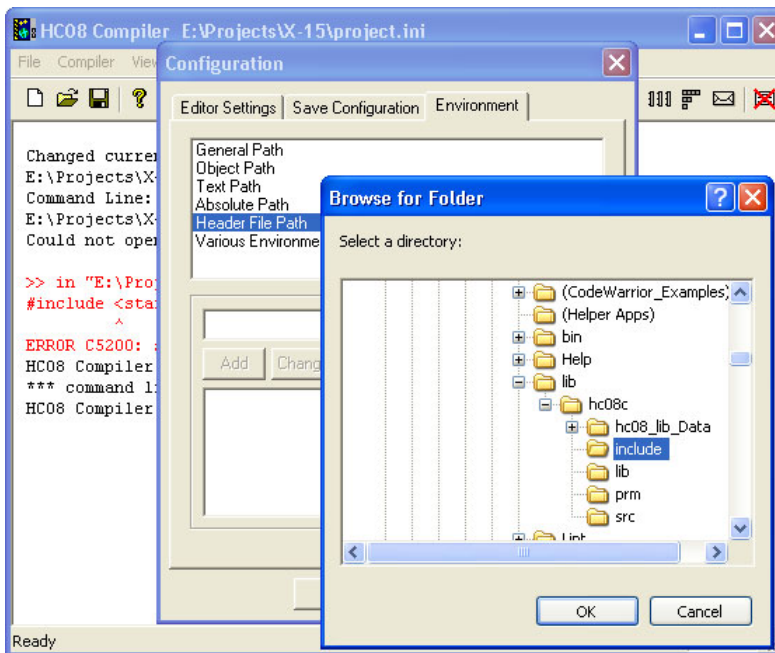
---

**NOTE** If you read the `Start08.c` file, you could have anticipated this on account of the `#include` preprocessor directive on line 21 for this header file.

---

The Compiler needs a configurational modification so that it can locate any missing files. Select *File > Configuration...* The *Configuration* dialog box appears (Figure 1.25).

**Figure 1.25** Browsing for the include subfolder in the CodeWarrior lib folder



Select the *Environment* tab in the *Configuration* dialog box and then select *Header File Path*.

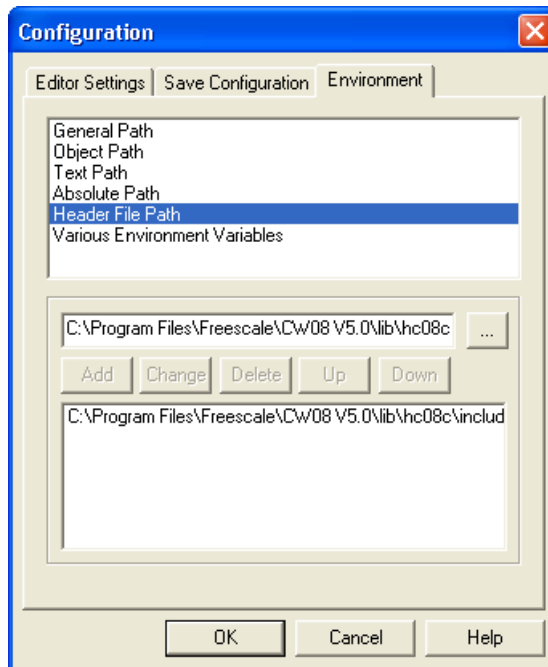
---

**NOTE** The environment variable associated with the *Header File Path* is the `LIBPATH` or `LIBRARYPATH`: '`include <File>`' Path variable. There is a hierarchy level for the order in which the Compiler searches for files. Confer "Files used with the Compiler" on page 133.

---

Press the “...” button and navigate in the *Browse for Folder* dialog box for the folder that contains the missing file - the `include` subfolder in the CodeWarrior installation’s `lib` folder. Press *OK* to close the *Browse for Folder* dialog box. The *Configuration* dialog box is now again active (Figure 1.26).

**Figure 1.26** Adding a Header File Path



Press the *Add* button. The path to the header files “*C:\Program Files\Freescale\CW for HC08 V5.0\lib\hc12c\include*” now appears in the lower panel. Press *OK*. An asterisk now appears in the *Configuration Title* bar, so save the modification to the configuration by pressing the *Save* button or by *File > Save Configuration*. If you do not save the configuration, the Compiler will revert to last-saved configuration the next time the project is reopened. The asterisk disappears.

---

**TIP** You can clear the messages in the Compiler window at any time by selecting *View > Log > Clear Log*.

---

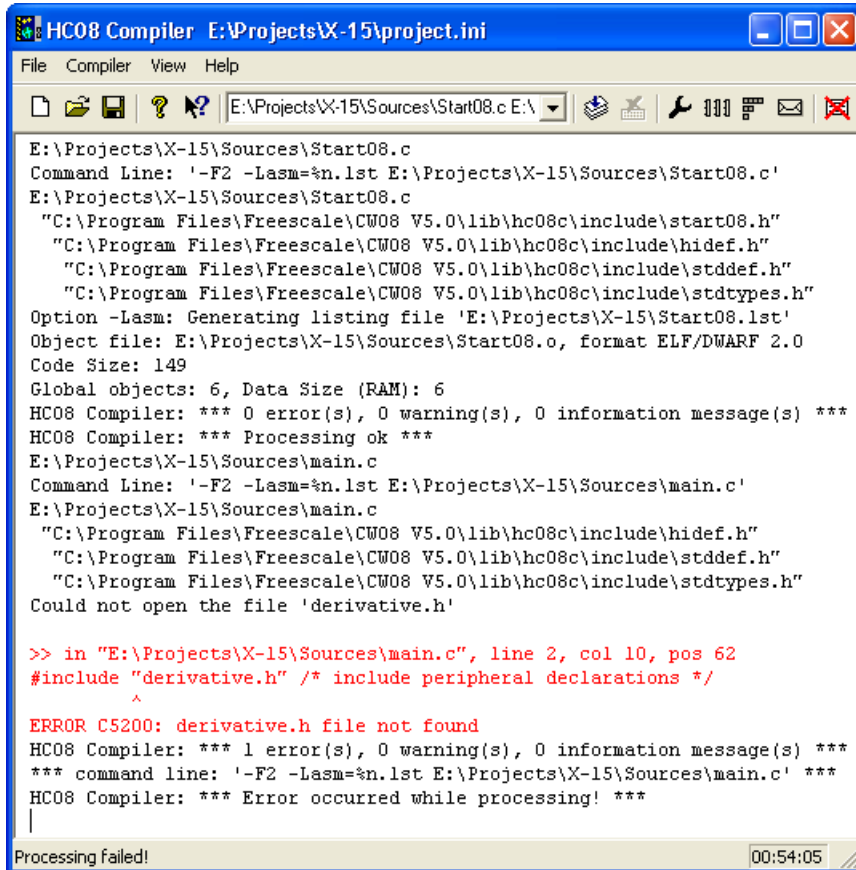
Now that you have supplied the path to the erstwhile missing file, you can try again to compile the `start08.c` file. Instead of compiling each file separately, you can compile any or all of them simultaneously.

Select *File > Compile* and again navigate to the Sources folder (in case it is not already active) and this time select both of the `*.c` files and press *Open* (Figure 1.27).

## Introduction

### Compilation with the Compiler

Figure 1.27 Partial compilation - one object file created...



```
E:\Projects\X-15\Sources\Start08.c
Command Line: '-F2 -Lasm=%n.lst E:\Projects\X-15\Sources\Start08.c'
E:\Projects\X-15\Sources\Start08.c
"C:\Program Files\Freescale\CW08 V5.0\lib\hc08c\include\start08.h"
"C:\Program Files\Freescale\CW08 V5.0\lib\hc08c\include\hidef.h"
"C:\Program Files\Freescale\CW08 V5.0\lib\hc08c\include\stddef.h"
"C:\Program Files\Freescale\CW08 V5.0\lib\hc08c\include\stdtypes.h"
Option -Lasm: Generating listing file 'E:\Projects\X-15\Start08.lst'
Object file: E:\Projects\X-15\Sources\Start08.o, format ELF/DWARF 2.0
Code Size: 149
Global objects: 6, Data Size (RAM): 6
HC08 Compiler: *** 0 error(s), 0 warning(s), 0 information message(s) ***
HC08 Compiler: *** Processing ok ***
E:\Projects\X-15\Sources\main.c
Command Line: '-F2 -Lasm=%n.lst E:\Projects\X-15\Sources\main.c'
E:\Projects\X-15\Sources\main.c
"C:\Program Files\Freescale\CW08 V5.0\lib\hc08c\include\hidef.h"
"C:\Program Files\Freescale\CW08 V5.0\lib\hc08c\include\stddef.h"
"C:\Program Files\Freescale\CW08 V5.0\lib\hc08c\include\stdtypes.h"
Could not open the file 'derivative.h'

>> in "E:\Projects\X-15\Sources\main.c", line 2, col 10, pos 62
#include "derivative.h" /* include peripheral declarations */
    ^
ERROR C5200: derivative.h file not found
HC08 Compiler: *** 1 error(s), 0 warning(s), 0 information message(s) ***
*** command line: '-F2 -Lasm=%n.lst E:\Projects\X-15\Sources\main.c' ***
HC08 Compiler: *** Error occurred while processing! ***
|
Processing failed! 00:54:05
```

The Compiler indicates successful compilation for the `Start08.c` file and displays some positive feedback for this file:

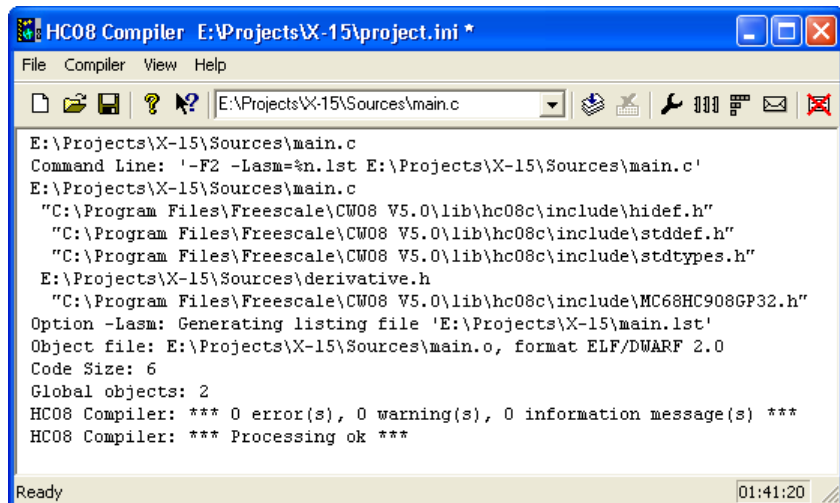
- Four header files were included either by the `Start08.c` file or one or more header files.
- An assembler listing file was generated for the `Start08.c` file - `Start08.lst`.
- An object file was generated in the `Sources` folder - `Start08.o` - using the ELF/DWARF 2.0 format.
- The `Code Size` was 149 bytes.
- Six global objects were created and the `Data Size` was 6 bytes in RAM.
- There were 0 errors, 0 warnings, and 0 information messages.

However, things did not go so well for the other C-source file - `main.c`. The Compiler was only able to locate three of the four header files it needs. It could not find the `derivative.h` header file. Another C5200 error message was generated. Therefore, you have to locate and make the path for the missing file known to the Compiler.

The `derivative.h` file is in the `Sources` folder, so you have to go through the routine again for adding another header path. Select *File > Configuration... > Environment > Header File Path*. In the *Browse for Folder* dialog box, browse for and select the `Sources` folder. Click *OK* to close the *Browse for Folder* dialog box. Press *Add* in the *Browse for Folder* dialog box and press *OK* to close this dialog box.

Unless there is another “missing” header file included by the `derivative.h` file, you are now ready to compile the `main.c` file. From the *File* menu, select *Compile* and select the `main.c` file in the `Sources` folder. (No harm is done if an already compiled C-source file is also selected.) Press *Open* to start the compilation process (Figure 1.28).

**Figure 1.28 Successful compilation of the main.c file...**



```
HC08 Compiler E:\Projects\X-15\project.ini *
File Compiler View Help
E:\Projects\X-15\Sources\main.c
E:\Projects\X-15\Sources\main.c
Command Line: '-F2 -Lasm=%n.lst E:\Projects\X-15\Sources\main.c'
E:\Projects\X-15\Sources\main.c
"C:\Program Files\Freescale\CW08 V5.0\lib\hc08c\include\hidef.h"
"C:\Program Files\Freescale\CW08 V5.0\lib\hc08c\include\stddef.h"
"C:\Program Files\Freescale\CW08 V5.0\lib\hc08c\include\stdtypes.h"
E:\Projects\X-15\Sources\derivative.h
"C:\Program Files\Freescale\CW08 V5.0\lib\hc08c\include\MC68HC908GP32.h"
Option -Lasm: Generating listing file 'E:\Projects\X-15\main.lst'
Object file: E:\Projects\X-15\Sources\main.o, format ELF/DWARF 2.0
Code Size: 6
Global objects: 2
HC08 Compiler: *** 0 error(s), 0 warning(s), 0 information message(s) ***
HC08 Compiler: *** Processing ok ***
Ready 01:41:20
```

You already know how to read the Compiler window after a compilation. The message “\*\*\* 0 error(s),” indicates that the `main.c` file compiled without errors. There is an asterisk in the Title bar again, so do not forget to save the configuration one additional time.

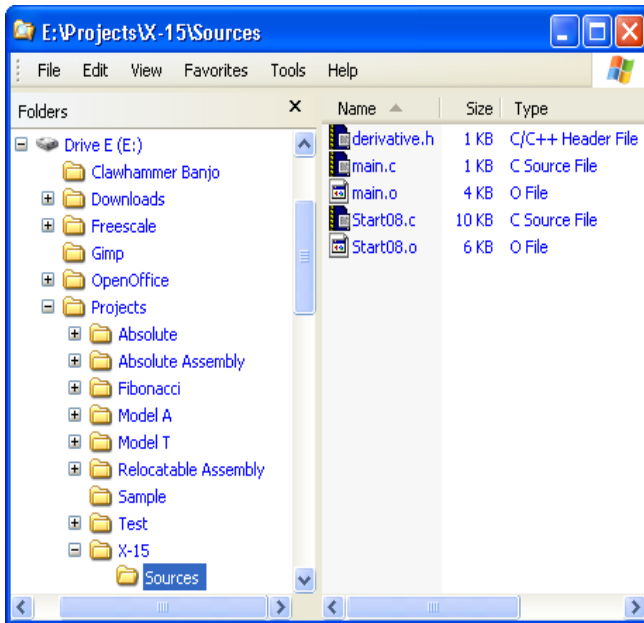
The Compiler generated object files in the same location where the C-source files are located - the `Sources` folder - (for further processing by the Linker), and output listing files were generated in the project directory. The binary object files have the same names as the input modules, but with the ‘\*.o’ extension instead. The assembly output files for each C-source file is similarly named (Figure 1.29).

## Introduction

### Compilation with the Compiler

---

Figure 1.29 Project directory after successful compilation



However, you are not yet finished. Only two of the three C-source files have been compiled. The remaining C-source file - MC68HC908GP32.C - is located in the lib folder in the CodeWarrior installation:

```
<CodeWarrior installation>\lib\HC08c\src
```

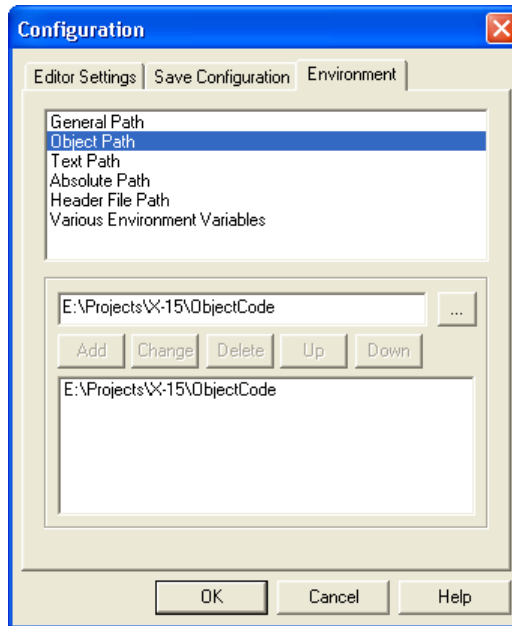
We recommend that you NOT generate object-code files for your projects in the CodeWarrior lib folder. Unless you specify where the object-code files are to be generated, they will be generated in the same folder that contained the C-source code files.

Using the Windows Explorer, create a new folder in the project directory and name it accordingly, say ObjectCode - the same name that the CodeWarrior uses for this purpose. You will generate the object code for the CodeWarrior library C-source file - MC68HC908GP32.C - in this new folder.

We use another environment variable for altering the path for an object code file - "OBJPATH: Object File Path" on page 125. Select *File > Configuration... > Environment > Object Path*. Navigate to the new folder for placing the object code in the project directory and press *OK*. Add to add it to the lower panel \*().

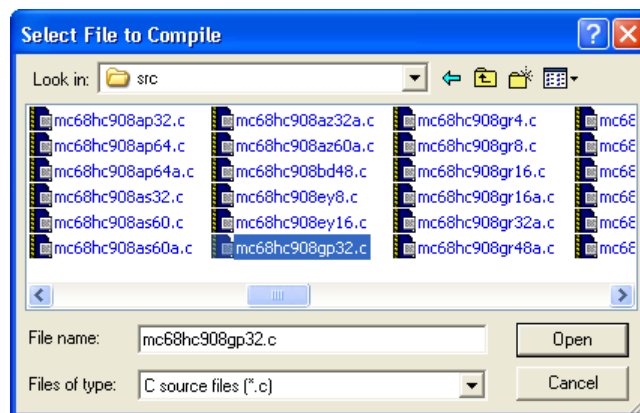


Figure 1.30 Adding an OBJPATH



Press *OK* to close the *Configuration* dialog box. Select *File > Compile* and browse for the C-source file in the `src` subfolder in the `lib` folder (Figure 1.31).

Figure 1.31 Compiling a lib source file...



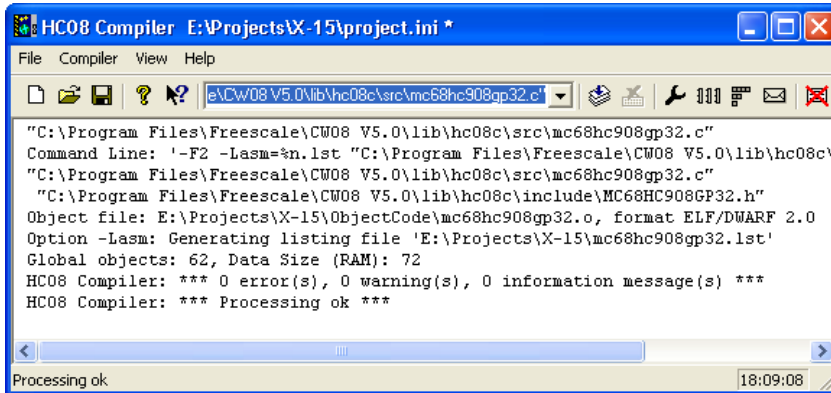
## Introduction

### Application Programs (Build Tools)

---

Press *Open* and the final C-source file should compile OK. And it does (Figure 1.32).

**Figure 1.32** C-source file in src folder is now compiled...



The Compiler log states that the `mc68hc908gp32.o` object-code file was created in the `ObjectCode` folder, as planned. Save the *Configuration* again in case you may want to recompile any of the C-source files in the future. Also move the other two object code files to the `ObjectCode` folder so that they are all located in the same location. This makes it easier to manage your project.

The haphazard running of this project was intentionally designed to fail in order to illustrate what would occur if the path of any header file is not properly configured. Be aware that header files may be included by C-source or other header files. The `lib` folder in the CodeWarrior installation contains several derivative-specific header and other files available for inclusion into your projects.

When you create another project with the Build Tool Utilities, plan ahead and set up in advance the Configurations for any input and output files.

Now that the project's object code files are available, the Linker Build Tool (`linker.exe`) together with an appropriate `*.prm` file for the CPU-derivative used in the project could link these object-code files together with any necessary library files to create a `*.abs` executable output file. See the *Linker section in the Build Tool Utilities manual* for details. However, using the CodeWarrior Development Studio is much faster and easier to set up or configure for this purpose.

## Application Programs (Build Tools)

You can find the standalone application programs (Build Tools) in the `\prog` directory where you installed the CodeWarrior software. For example, if you installed the CodeWarrior software in the `C:\Freescale` directory, all the Build Tools are located in the `C:\Freescale\prog` directory with the exception of the CodeWarrior IDE.exe file, which is found in the `bin` subfolder of the CodeWarrior installation.

The following list is an overview of the tools used for C programming:

- `IDE.exe` - CodeWarrior IDE

- `chc08.exe` - Freescale HC08 Compiler
- `ahc08.exe` - Freescale HC08 Assembler
- `libmaker.exe` - Librarian Tool to build libraries
- `linker.exe` - Link Tool to build applications (absolute files)
- `decoder.exe` - Decoder Tool to generate assembly listings
- `maker.exe` - Make Tool to rebuild automatically
- `burner.exe` - Batch and interactive Burner (S-Records, ...)
- `hiwave.exe` - Multi-Purpose Simulation or Debugging Environment
- `pipec.exe` - Utility to redirect messages to `stdout`

---

**NOTE** Depending on your license configuration, not all programs listed above may be installed or there might be additional programs.

---

## Startup Command-Line Options

There are some special tool options. These tools are specified at tool startup (while launching the tool). They cannot be specified interactively:

- `-Prod`: Specify Project File at Startup specifies the current project directory or file (Listing 1.2).

### Listing 1.2 An example of a startup command-line option

---

```
linker.exe -Prod=C:\Freescale\demo\myproject.pjt
```

---

There are other options that launch the tool and open its special dialog boxes. Those dialog boxes are available in the compiler, assembler, burner, maker, linker, decoder, or libmaker:

- `ShowOptionDialog`: This startup option (see Listing 1.3) opens the tool's option dialog box.
- `ShowMessageDialog`: This startup option opens the tool message dialog box.
- `ShowConfigurationDialog`: This opens the *File->Configuration* dialog box.
- `ShowBurnerDialog`: This option is for the Burner only and opens the Burner dialog box.
- `ShowSmartSliderDialog`: This option is for the compiler only and opens the smart slider dialog box.
- `ShowAboutDialog`: This option opens the tool about box.

## Introduction

### Highlights

---

The above options open a modal dialog box where you can specify tool settings. If you press the OK button of the dialog box, the settings are stored in the current project settings file.

#### Listing 1.3 An example of storing options in the current project settings file

---

```
C:\Freescale\prog\linker.exe -ShowOptionDialog  
                               -Prod=C:\demos\myproject.pjt
```

---

## Highlights

- Powerful User Interface
- Online Help
- Flexible Type Management
- Flexible Message Management
- 32-bit Application
- Support for Encrypted Files
- High-Performance Optimizations
- Conforms to ANSI/ISO 9899-1990

## CodeWarrior Integration of the Build Tools

All required plug-ins are installed together with the CodeWarrior IDE. The CodeWarrior IDE is installed in the `bin` directory (usually `C:\CodeWarrior\bin`). The plug-ins are installed in the `bin\plugins` directory.

## Combined or Separated Installations

The installation script enables you to install several CPUs in one single installation path. This saves disk space and enables switching from one processor family to another without leaving the IDE.

---

**NOTE** In addition, it is possible to have separate installations on one machine. There is only one point to consider: The IDE uses COM files, and for COM the IDE installation path is written into the Windows Registry. This registration is done in the installation setup. However, if there is a problem with the COM registration using several installations on one machine, the COM registration is done by starting a small batch file located in the 'bin' (usually the

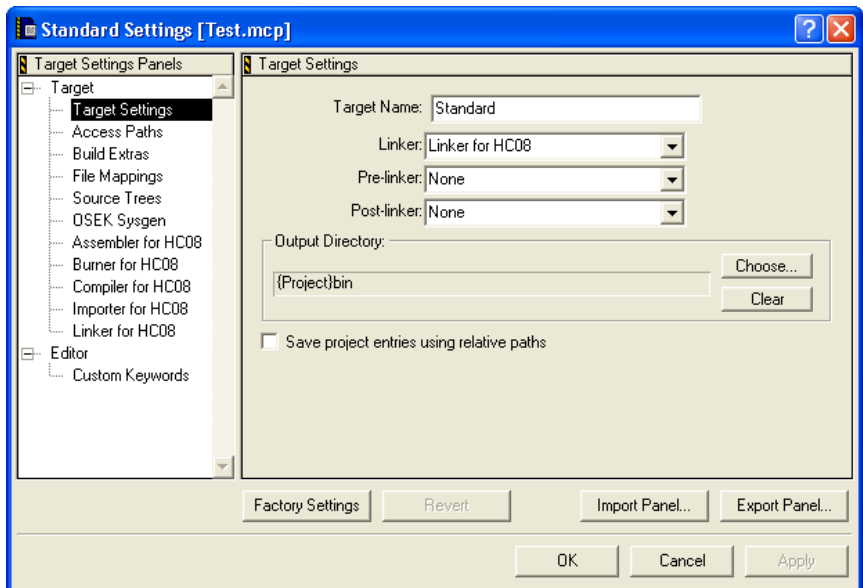
C:\CodeWarrior\bin) directory. To do this, start the `regservers.bat` batch file.

---

## Target Settings preference panel

The linker builds an absolute (\*.abs) file. Before working with a project, set up the linker for the selected CPU in the *Target Settings* preference panel (Figure 1.33).

**Figure 1.33** Target Settings preference panel



Depending on the CPU targets installed, you can select from the various linkers available in the Linker drop box.

You can add PC-lint functionality to an existing project by creating a new build target. In your project, select the *Targets* tab in the project window and select *Project > Create Target...* to display the *New Target* dialog box (Figure 1.34).

## Introduction

### CodeWarrior Integration of the Build Tools

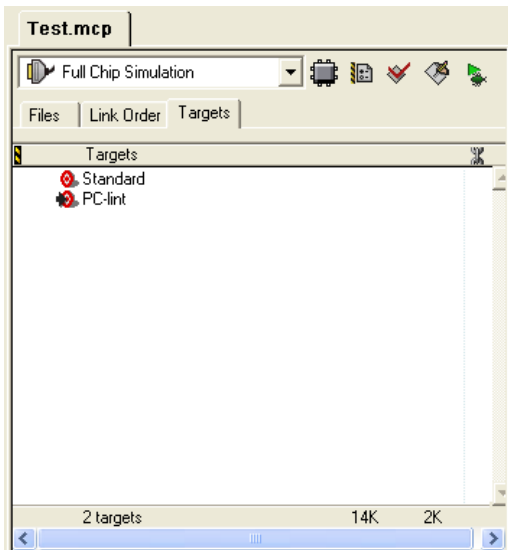
---

Figure 1.34 New Target dialog box



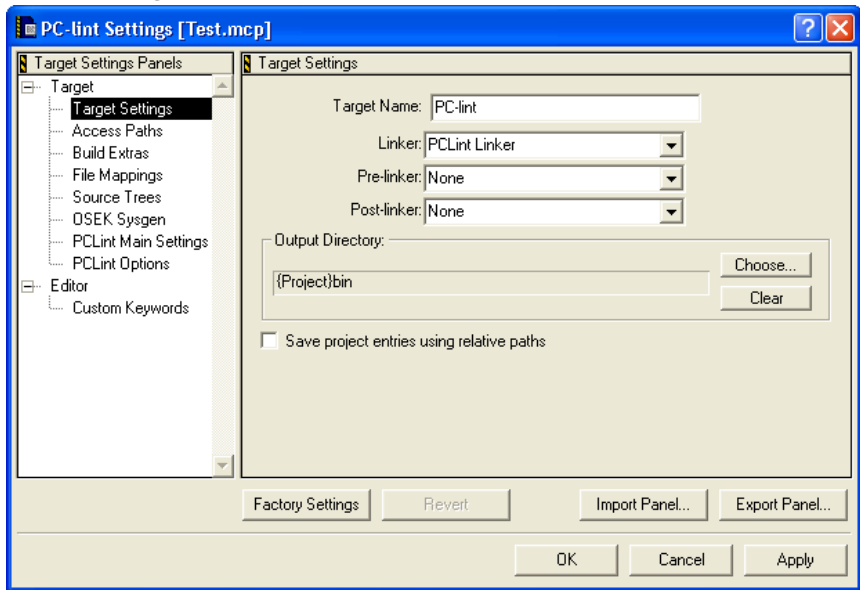
Enter a name, say *PC-lint*, in the *Name for new target:* text box and select the *Clone existing target* option and click *OK*. The new build target appears in the project window (Figure 1.35).

Figure 1.35 New build target



Select the new *PC-lint* target that appears in the *Targets* panel and select *Edit > PC-lint Settings...* (Figure 1.36).

Figure 1.36 PC-lint Settings preference panel



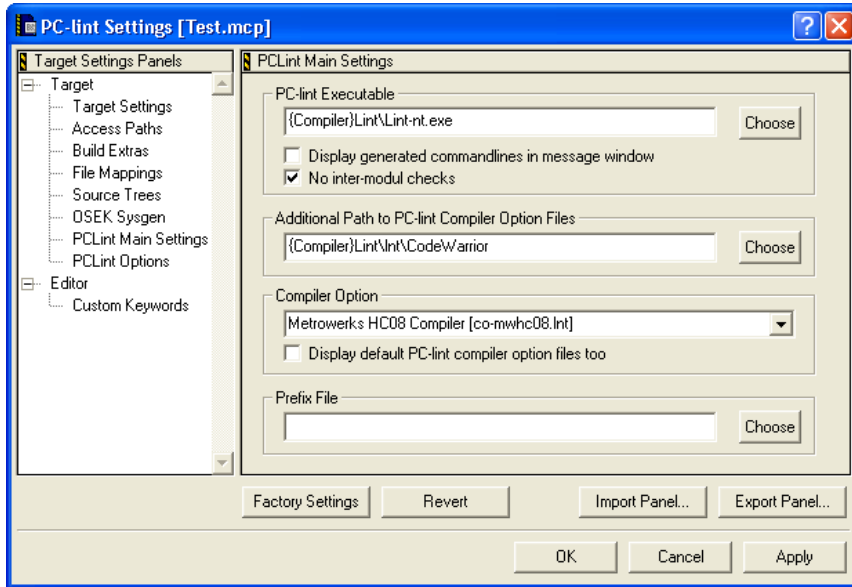
In the *Target Settings* preference panel, click on the dropdown menu for the *Linker:* and select *PCLint Linker*. Select *PCLint Main Settings* that appears in left column after *PCLint Linker* is selected (Figure 1.37).

## Introduction

### CodeWarrior Integration of the Build Tools

---

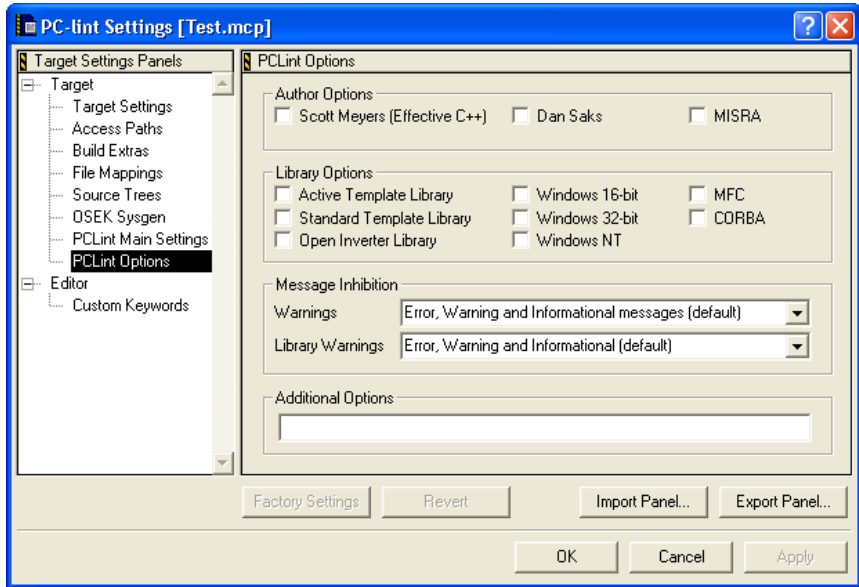
Figure 1.37 PCLint Main Settings preference panel



In the *PCLint Main Settings* preference panel make sure the PC-lint executable path is correct and select *HC08 Compiler* from the *Compiler Option* dropdown menu. Click *OK*. You can also set the options in the *PCLint Options* preference panel (Figure 1.38). See the documentation from Gimpel Software for detailed directions.



Figure 1.38 PCLint Options preference panel



You can also select a libmaker. When a libmaker is set up, the build target is a library (\* .lib) file. Furthermore, you may decide to rename the project's target by entering its name in the *Target Name*: text box.

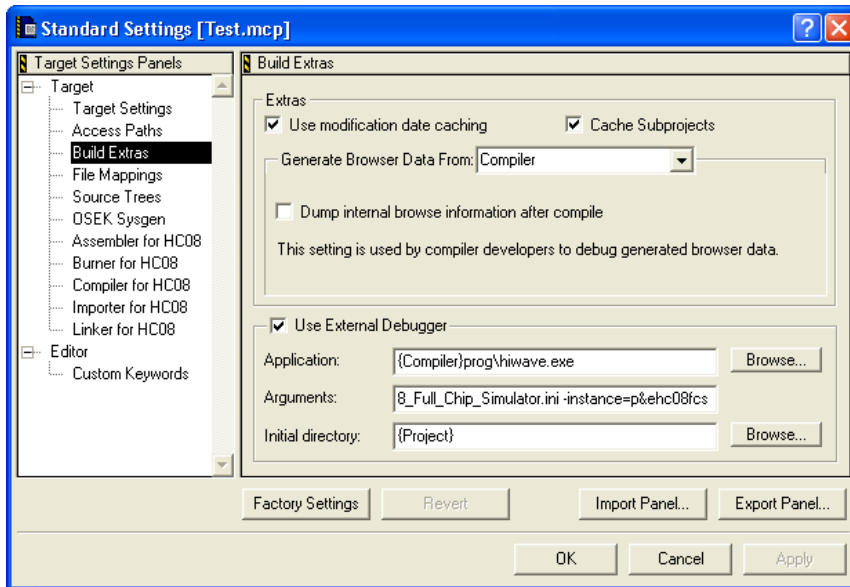
## Build Extras preference panel

See Figure 1.39 for the *Build Extras* preference panel.

## Introduction

### CodeWarrior Integration of the Build Tools

Figure 1.39 Build Extras preference panel



Use the *Build Extras* preference panel so the compiler can generate browser information. Enable the *Use External Debugger*: check box to use the external simulator or debugger. Define the path to the debugger, either:

- absolute  
for example, 'C:\Freescale\prog\hiwave.exe'
- installation-relative  
for example, '{Compiler}prog\hiwave.exe'

Additional command-line arguments passed to the debugger are specified in the *Arguments*: text box. In addition to the normal arguments (refer to your simulator or debugger documentation), the following '% macros' can also be specified:

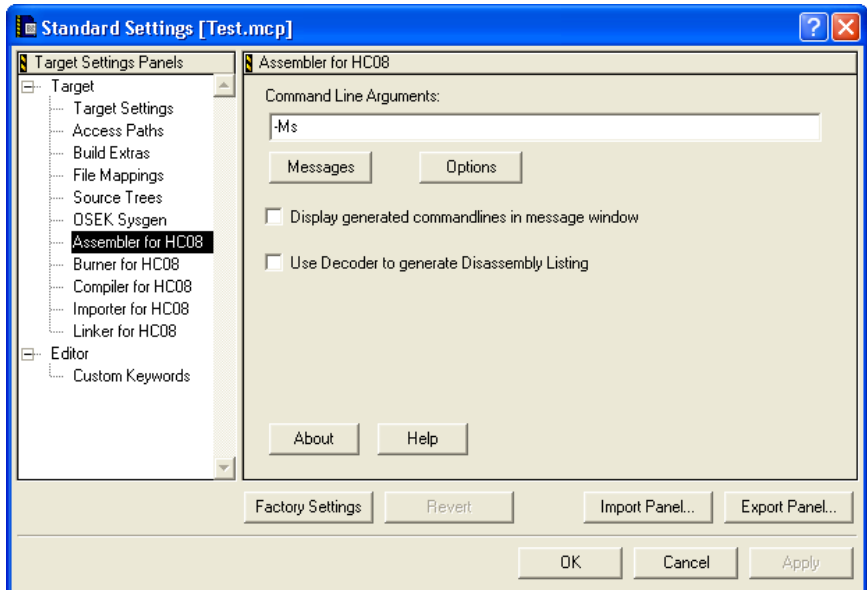
- %sourceFilePath
- %sourceFileDir
- %sourceFileName
- %sourceLineNumber
- %sourceSelection
- %sourceSelUpdate
- %projectFilePath

- %projectFileDir
- %projectFileName
- %projectSelectedFiles
- %targetFilePath
- %targetFileDir
- %targetFileName
- %currentTargetName
- %symFilePath
- %symFileDi
- %symFileName

## Assembler for HC08 preference panel

See Figure 1.40 for the Assembler for HC08 preference panel.

**Figure 1.40 Assembler for HC08 Preference Panel**



## Introduction

### *CodeWarrior Integration of the Build Tools*

---

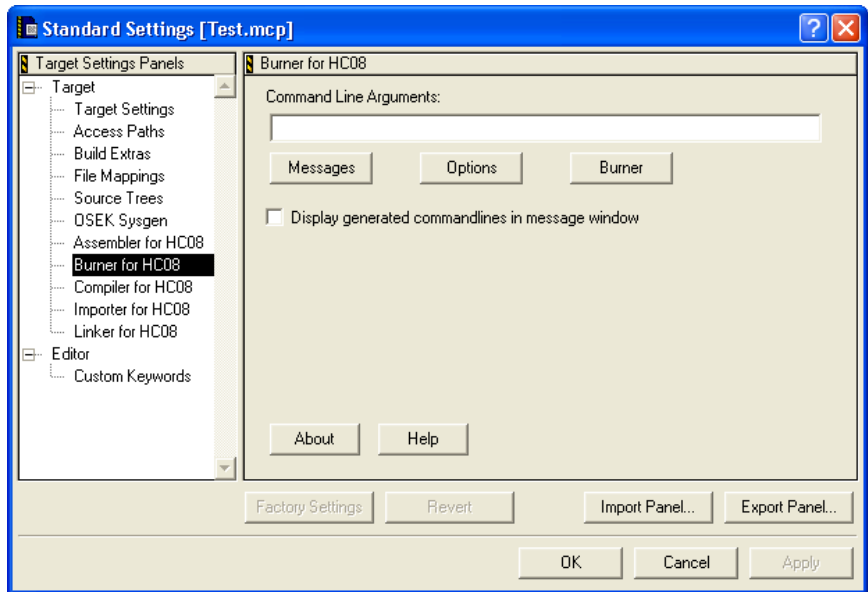
The *Assembler for HC08* preference panel contains the following:

- *Command Line Arguments*: Command-line options are displayed. You can add, delete, or modify the options by hand, or by using the Messages and Options buttons below.
  - *Messages*: Button to open the Messages dialog box
  - *Options*: Button to open the Options dialog box
- *Display generated commandlines in message window*: The plug-in filters the messages produced, so that only Warning, Information, or Error messages are displayed in the ‘Errors & Warnings’ window. With this check box set, the complete command line is passed to the tool.
- *Use Decoder to generate Disassembly Listing*: The built-in listing file generator is used to produce the disassembly listing. If this check box is set, the external decoder is enabled.
- *About*: Provides status and version information.
- *Help*: Opens the help file.

## Burner for HC08 preference panel

The Burner Plug-In Function: The \* .bb1 (batch burner language) files are mapped to the Burner Plug-In in the File Mappings Preference Panel. Whenever a \* .bb1 file is in the project file, the \* .bb1 file is processed during the post-link phase using the settings in the Burner preference panel (Figure 1.41).

Figure 1.41 Burner for HC08 preference Panel



The *Burner for HC08* preference panel contains the following:

- *Command Line Arguments*: The actual command line options are displayed. You can add, delete, or modify the options manually, or use the *Messages*, *Options*, and *Burner* buttons listed below.
  - *Messages*: Opens the *Messages* dialog box
  - *Options*: Opens the *Options* dialog box
  - *Burner*: Opens the *Burner* dialog box
- *Display generated commandlines in message window*: The plug-in filters the messages produced, so that only Warning, Information, or Error messages are displayed in the 'Errors & Warnings' window. With this check box set, the complete command line is passed to the tool.
- *About*: Provides status and version information.
- *Help*: Opens the help file.

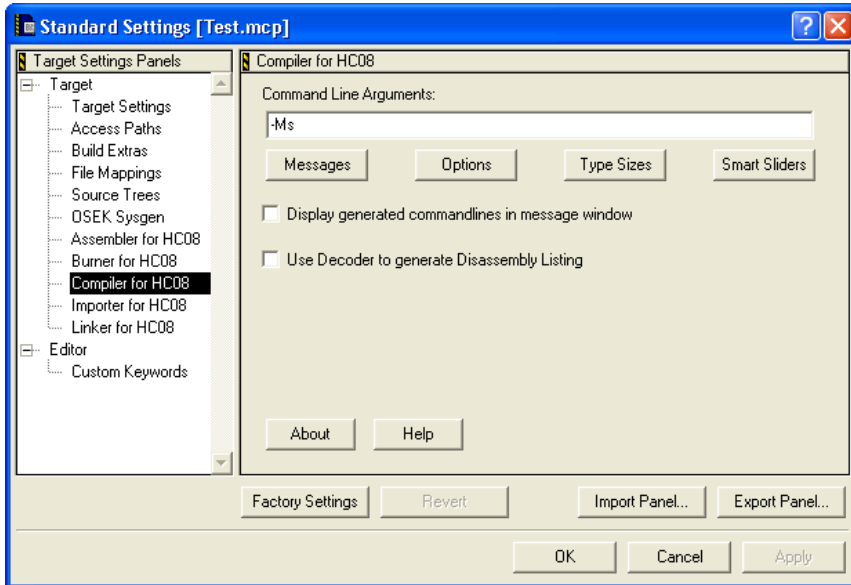
## Compiler for HC08 preference panel

The *Compiler for HC08* preference panel (Figure 1.42) contains the following:

## Introduction

### CodeWarrior Integration of the Build Tools

Figure 1.42 Compiler for HC08 preference panel

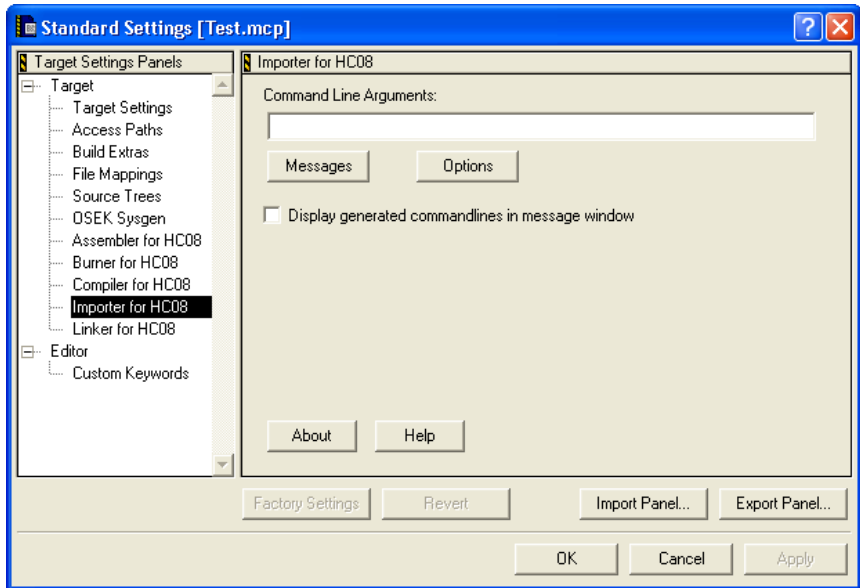


- *Command Line Arguments:* Command line options are displayed. You can add, delete, or modify the options manually, or use the *Messages*, *Options*, *Type Sizes*, and *Smart Sliders* buttons listed below.
  - *Messages:* Opens the *Messages* dialog box
  - *Options:* Opens the *Options* dialog box
  - *Type Sizes:* Opens the *Standard Type Size* dialog box
  - *Smart Sliders:* Opens the *Smart Slider* dialog box
- *Display generated commandlines in message window:* The plug-in filters the messages produced, so that only Warning, Information, or Error messages are displayed in the *Errors & Warnings* window. With this check box set, the complete command line is passed to the tool.
- *Use Decoder to generate Disassembly Listing:* Checking this check box enables the external decoder to generate a disassembly listing.
- *About:* Provides status and version information.
- *Help:* Opens the help file.

## Importer for HC08 preference panel

See Figure 1.43 for the *Importer for HC08* preference panel.

Figure 1.43 Importer for HC08 preference panel



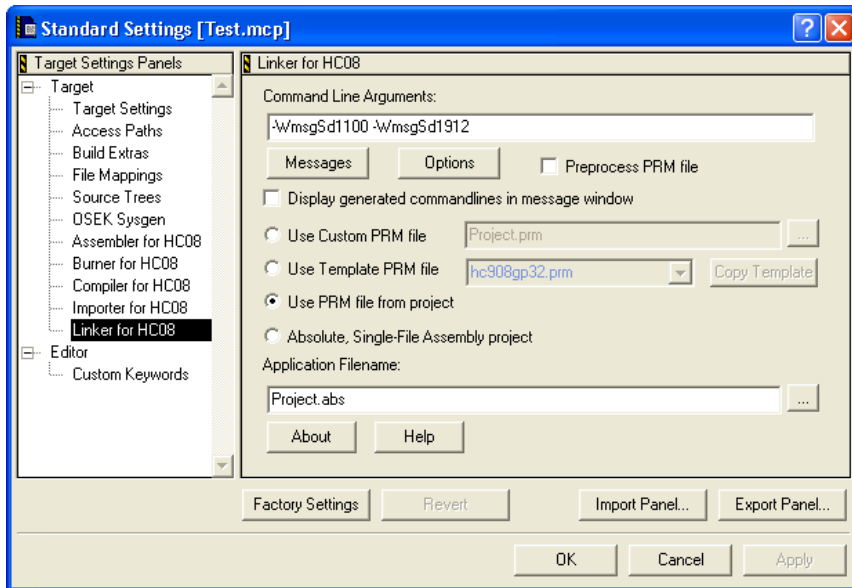
The *Importer for HC08* preference panel contains the following controls:

- *Command-line Arguments*: Command-line options are displayed. You can add, delete, or modify the options manually, or use the *Messages* or *Options* buttons listed below.
  - *Messages*: Opens the *Messages* dialog box
  - *Options*: Opens the *Options* dialog box
- *Display generated commandlines in message window*: The plug-in filters the messages produced so that only Warning, Information, or Error messages are displayed in the *Errors & Warnings* window. With this check box set, the complete command line is passed to the tool.
- *About*: Provides status and version information.
- *Help*: Opens the help file.

## Linker for HC08 preference panel

The *Linker for HC08* preference panel (Figure 1.44) displays in the *Target Settings Panels* panel if the Linker is selected.

**Figure 1.44** Linker for HC08 preference panel



The *Linker for HC08* preference panel contains the following controls:

- *Command-line Arguments*: Command-line options are displayed. You can add, delete, or modify the options manually, or use the *Messages* or *Options* buttons listed below.
  - *Messages*: Opens the *Messages* dialog box
  - *Options*: Opens the *Options* dialog box
- *Display generated commandlines in message window*: The plug-in filters the messages produced, so that only Warning, Information, or Error messages are displayed in the *Errors & Warnings* window. With this check box set, the complete command line is passed to the tool.
- *Use Custom PRM file*: Specifies a custom linker parameter file in the edit box. Use the browse button (...) to browse for a file.
- *Use Template PRM file*: With this radio control set, you can select one of the pre-made PRM files located in the templates directory (usually



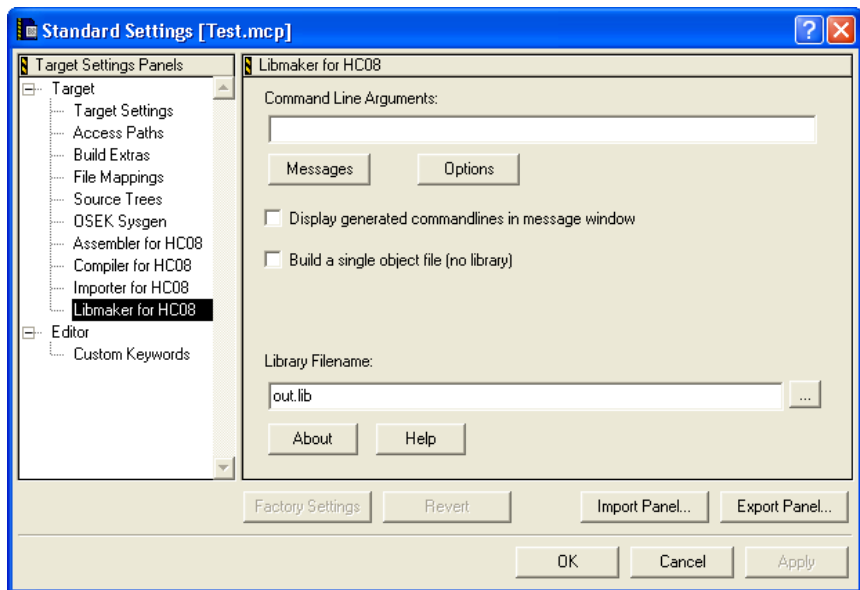
C:\Freescale\templates\<target>\prn). By employing the 'Copy Template' button, the user can copy a template PRM file into the project to maintain a local copy.

- *Application Filename*: The output filename is specified.
- *About*: Provides status and version information.
- *Help*: Button to open the tool help file directly.

## Libmaker for HC08 preference panel

The *Libmaker for HC08* panel (Figure 1.45) displays in the *Target Settings Panels* panel if a Libmaker is selected.

**Figure 1.45** Libmaker for HC08 preference panel



The *Libmaker for HC08* preference panel contains the following:

- *Command-line Arguments*: Command-line options are displayed. You can add, delete, or modify the options manually, or by using the *Messages* or *Options* buttons listed below.
  - *Messages*: Opens the *Messages* dialog box
  - *Options*: Opens the *Options* dialog box

## Introduction

### Integration into Microsoft Visual Studio (Visual C++ V5.0 or later)

---

- *Display generated commandlines in message window:* The plug-in filters the messages produced, so that only Warning, Information, or Error messages are displayed in the *Errors & Warnings* window. With this check box set, the complete command line is passed to the tool.
- *Build a single object file (no library):* With this check box set, only a single object file is generated. This is useful to generate a startup object file to be linked later, or to create an absolute assembly application.
- *Library Filename:* The output filename is specified. Use the browse button (directly to the right of the entry field) to locate the directory where the output file is stored.
- *About:* Provides status and version information.
- *Help:* Opens the help file.

## CodeWarrior Tips and Tricks

If the Simulator or Debugger cannot be launched, check the settings in the *Build Extras* preference panel.

If the data folder of the project is deleted, then some project-related settings may also have been deleted. One of these settings determines whether the debugger or simulator is enabled. Check the menu *Project > Enable/Disable Debugger*.

If a file cannot be added to the project, its file extension may be absent from the *File Mappings* preference panel. Adding this file's extension to the listing in the *File Mappings* preference panel should correct this.

If it is suspected that project data is corrupted, export and re-import the project using *File->Export Project* and *File->Import Project*.

## Integration into Microsoft Visual Studio (Visual C++ V5.0 or later)

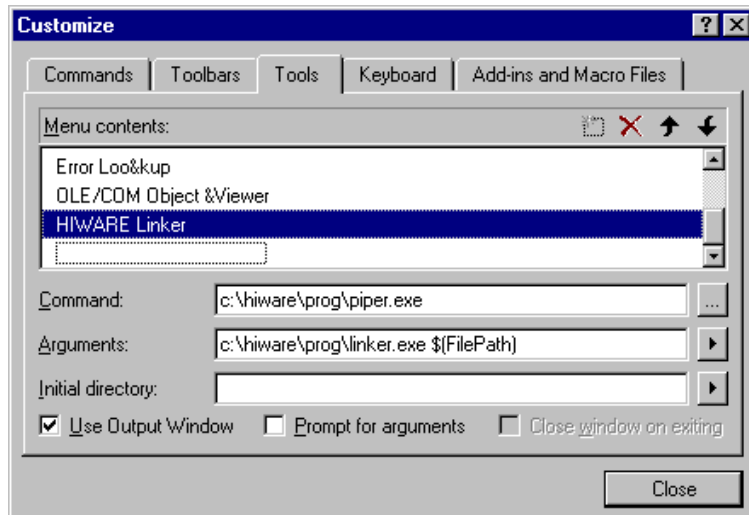
Use the following procedure to integrate the Tools into the Microsoft Visual Studio (Visual C++).

### Integration as Additional Tools

1. Start Visual Studio.
2. Select the menu *Tools->Customize*.
3. Select the *'Tools'* Tab.
4. Add a new tool using the *'New'* button, or by double-clicking on the last empty entry in the *'Menu contents'* list.
5. Type in the name of the tool to display in the menu (for example, *'Linker'*).

6. In the 'Command' field, type in the name and path of the piper tool (for example, 'C:\Freescale\prog\piper.exe'.
7. In the 'Arguments' field, type in the name of the tool to be started with any command line options (for example, -N) and the \$(FilePath) Visual variable (for example, 'C:\Freescale\prog\linker.exe -N \$(FilePath)').
8. Check 'Use Output Window'.
9. Uncheck 'Prompt for arguments'.
10. Proceed as above for all other tools (for example, compiler, decoder).
11. Close the 'Customize' dialog box (Figure 1.46).

**Figure 1.46 Customize dialog box**



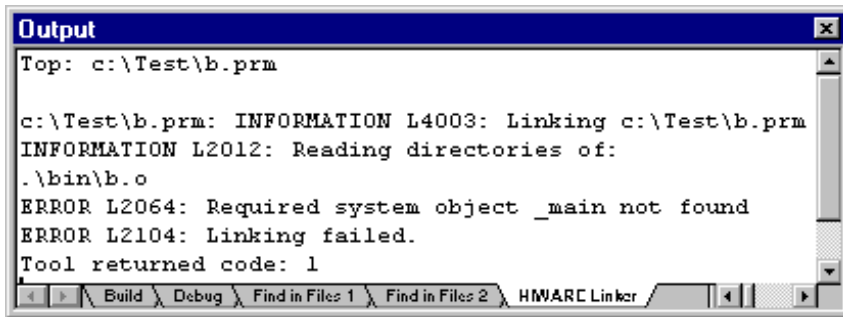
This allows the active file to be compiled or linked in the Visual Editor ('\$(FilePath)'). Tool messages are reported in a separate Visual output window (Figure 1.47). Double click on the output entries to jump to the right message position (message feedback).

## Introduction

Integration into Microsoft Visual Studio (Visual C++ V5.0 or later)

---

Figure 1.47 Visual output window



```
Output
Top: c:\Test\b.prm

c:\Test\b.prm: INFORMATION L4003: Linking c:\Test\b.prm
INFORMATION L2012: Reading directories of:
.\bin\b.o
ERROR L2064: Required system object _main not found
ERROR L2104: Linking failed.
Tool returned code: 1

Build Debug Find in Files 1 Find in Files 2 HWARC Linker
```

## Integration with Visual Studio Toolbar

1. Start Visual Studio.
2. Make sure that all tools are integrated as *Additional Tools*.
3. Select the menu *Tools > Customize*.
4. Select the *Toolbars* tab.
5. Select *New* and enter a name (for example, *Freescale Tools*). A new empty toolbar named '*Freescale Tools*' appears on your screen.
6. Select the *Commands* tab.
7. In the *Category* drop down box, select *Tools*.
8. On the right side many 'hammer' tool images appear, each with a number. The number corresponds to the entry in the Tool menu. Usually the first user-defined tool is tool number 7. (The Linker was set up in *Additional Tools* above.)
9. Drag the selected tool image to the *Freescale Tools* toolbar.
10. All default tool images look the same, making it difficult to know which tool has been launched. You should associate a name with them.
11. Right-click on a tool in the *Freescale Tools* toolbar to open the context menu of the button.
12. Select *Button Appearance...* in the context menu.
13. Select *Image and Text*.
14. Enter the tool name to associate with the image in '*Button text:*' (for example, '*Linker*').
15. Repeat for all tools to appear in the toolbar.
16. Close the *Customize* dialog box.

This enables the tools to be started from the toolbar.

The Compiler provides the following language settings:

- ANSI-C: The compiler can behave as an ANSI-C compiler. It is possible:
  - to force the compiler into a strict ANSI-C compliant mode, or
  - to use language extensions that are specially designed for more efficient embedded systems programming.

## Object-File Formats

The Compiler supports two different object-file formats: ELF/DWARF and the vendor-specific HIWARE object-file format. The object-file format specifies the format of the object files (\*.o extension), the library files (\*.lib extension), and the absolute files (\*.abs extension).

---

**NOTE** Be careful and do not mix object-file formats. *Both the HIWARE and the ELF/DWARF object files use the same filename extensions.*

---

### HIWARE Object-File Format

The HIWARE Object-File Format is a vendor-specific object-file format defined by HIWARE AG. This object-file format is very compact. The object file sizes are smaller than the ELF/DWARF object files. This smaller size enables faster file operations on the object files. The object-file format is also easy to support by other tool vendors (for example, emulators from Abatron, Lauterbach, or iSYSTEM). The object-file format supports ANSI-C and Modula-2.

Each other tool vendor must support this object-file format explicitly. Note that there is also a lack of extensibility, amount of debug information, and C++ support. For example, using the full flexibility of the Compiler Type Management is not supported in the HIWARE Object-file Format.

Using the HIWARE object-file format may also result in slower source or debug info loading. In the HIWARE object-file format, the source position information is provided as position information (offset in file), and not directly in a file, line, or column format. The debugger must translate this HIWARE object-file source information format into a file, line, or column format. This has the tendency to slow down the source file or debugging info loading process.

### ELF/DWARF Object-File Format

The ELF/DWARF object-file format originally comes from the UNIX world. This format is very flexible and is able to support extensions.

Many chip vendors define this object-file format as the standard for tool vendors supporting their devices. This standard allows inter-tool operability making it possible to use the compiler from one tool vendor, and the linker from another. The developer has the choice to select the best tool in the tool chain. In addition, other third parties (for example, emulator vendors) only have to support this object file to support a wide range of tool vendors.

Object-file sizes are large compared with the HIWARE object-file format.

## Introduction

### Object-File Formats

---

---

**NOTE** ANSI-C and Modula-2 are supported in this object-file format.

---

## Tools

The CodeWarrior Suite contains the following Tools, among others:

### Compiler

The same Compiler executable supports both object-file formats. Use the -F (-Fh, -F1, -F1o, -F2, -F2o, -F6, or -F7): Object-File Format compiler option to switch the object-file format.

Note that not all Compiler backends support both ELF/DWARF and the HIWARE Object-File formats. Some only support one of the two.

### Decoder

Use the same executable 'decoder.exe' for both the HIWARE and the ELF/DWARF object-file formats.

### Linker

Use the same executable 'linker.exe' for both the HIWARE and the ELF/DWARF object-file formats.

### Simulator or Debugger

The Simulator or Debugger supports both object-file formats.

## Mixing Object-File Formats

Mixing HIWARE and ELF object files is not possible. Mixing ELF object files with DWARF 1.1 and DWARF 2.0 debug information is possible. However, the final generated application does not contain any debug data.

# Graphical User Interface

---

The Graphical User Interface (GUI) tool provides both a simple and a powerful user interface:

- Graphical User Interface
- Command-Line User Interface
- Online Help
- Error Feedback
- Easy integration into other tools (for example, CodeWarrior, CodeWright, MS Visual Studio, WinEdit, ...)

This chapter describes the user interface and provides useful hints. Its major elements are:

- Launching the Compiler
- Tip of the Day
- Main Window
- Window Title
- Content Area
- Toolbar
- Status Bar
- Menu Bar
- Standard Types dialog box
- Option Settings dialog box
- Compiler Smart Control dialog box
- Message Settings dialog box
- About... dialog box

## Launching the Compiler

Start the compiler using:

- The Windows Explorer
- An Icon on the desktop

## Graphical User Interface

### Launching the Compiler

---

- An Icon in a program group
- Batch and command files
- Other tools (Editor, Visual Studio, etc.)

## Interactive Mode

If the compiler is started with no input (that means no options and no input files), then the graphical user interface (GUI) is active (interactive mode). This is usually the case if the compiler is started using the Explorer or using an Icon.

## Batch Mode

If the compiler is started with arguments (options and/or input files), then it is started in batch mode (Listing 2.1).

### Listing 2.1 Specify the line associated with an icon on the desktop.

---

```
C:\Freescale\prog\chc08.exe -F2 a.c d.c
```

---

In batch mode, the compiler does not open a window. It is displayed in the taskbar only for the time it processes the input and terminates afterwards (Listing 2.2).

### Listing 2.2 Commands are entered to run as shown below.

---

```
C:\> C:\Freescale\prog\chc08.exe -F2 a.c d.c
```

---

Message output (stdout) of the compiler is redirected using the normal redirection operators (for example, '`>`' to write the message output to a file), as shown in Listing 2.3:

### Listing 2.3 Command-line message output is redirected to a file.

---

```
C:\> C:\Freescale\prog\chc08.exe -F2 a.c d.c > myoutput.o
```

---

The command line process returns after starting the compiling process. It does not wait until the started process has terminated. To start a process and wait for termination (for example, for synchronization), use the '`start`' command under Windows NT/95/98/Me/2000/XP, or use the '`/wait`' options (see windows help '`help start`'). Using '`start /wait`' (Listing 2.4) you can write perfect batch files (for example, to process your files).



---

**Listing 2.4 Start a compilation process and wait for termination**

---

```
C:\> start /wait C:\Freescale\prog\chc08.exe -F2 a.c d.c
```

---

## Tip of the Day

When the application is started, a standard *Tip of the Day* (Figure 2.1) window is opened containing the last news and tips.

The *Next Tip* button displays the next tip about the application.

If it is not desired for the *Tip of the Day* window to open automatically when the application is started, uncheck the check box *Show Tips on StartUp*.

---

**NOTE** This configuration entry is stored in the local project file.

---

To enable automatic display from the standard *Tip of the Day* window when the application is started, select the entry *Help | Tip of the Day...*. The *Tip of the Day* window will be open. Check the box *Show Tips on StartUp*.

Click *Close* to close the *Tip of the Day* window.

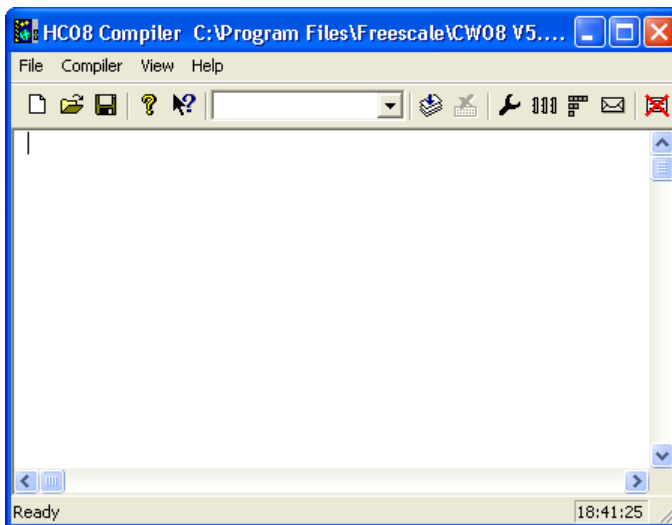
**Figure 2.1 Tip of the Day dialog box**



## Main Window

This Main Window (Figure 2.2) is empty when a filename is not specified while starting the application. The application window provides a window title, a menu bar, a toolbar, a content area, and a status bar.

**Figure 2.2** Main Window



## Window Title

The window title displays the application name and the project name. If there is no project currently loaded, the “Default Configuration” is displayed. An asterisk (\*) after the configuration name is present if any value has changed but has not yet been saved.

---

**NOTE** Changes to options, the Editor Configuration, and the application appearance can make the \* appear.

---

## Content Area

The content area is used as a text container, where logging information about the process session is displayed. This logging information consists of:

- The name of the file being processed
- The whole names (including full path specifications) of the files processed (main C file and all files included)
- An error, warning, and information message list
- The size of the code generated during the process session

When a file is dropped into the application window content area, the corresponding file is either loaded as configuration data or processed. It is loaded as configuration data if the file has the “\*.ini” extension. If the file does not contain this extension, the file is processed with the current option settings.

All text in the application window content area can contain context information. The context information consists of two items:

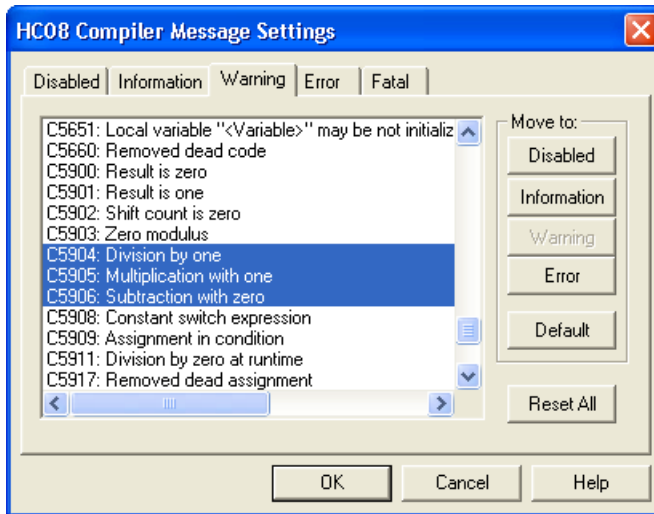
- A filename including a position inside of a file
- A message number

File context information is available for all output where a text file is considered. It is also available for all source and include files, and for messages which do concern a specific file. If a file context is available, double-clicking on the text or message opens this file in an editor, as specified in the Editor Configuration. The right mouse button can also be used to open a context menu. The context menu contains an “Open...” entry if a file context is available. If a file cannot be opened although a context menu entry is present, refer to Global Initialization File (mcutools.ini).

The message number is available for any message output. There are three ways to open the corresponding entry in the help file.

- Select one line of the message and *press F1*.  
If the selected line does not have a message number, the main help is displayed.
- Press *Shift-F1* and then click on the message text.  
If the point clicked at does not have a message number, the main help is displayed.
- Click with the right mouse at the message text and select “*Help on...*”.  
This entry is available only if a message number is available (Figure 2.3).

Figure 2.3 HC08 Compiler Message Settings dialog box



## Toolbar

The three buttons on the left in the Toolbar (Figure 2.4) are linked with the corresponding entries of the *File* menu. The next button opens the *About...* dialog box. After pressing the context help button (or the shortcut *Shift F1*), the mouse cursor changes its form and displays a question mark beside the arrow. The help file is called for the next item which is clicked. It is clicked on menus, toolbar buttons, and on the window area to get help specific for the selected topic.

Figure 2.4 Toolbar



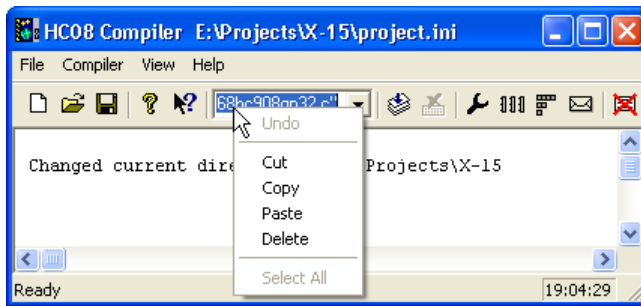
The command line history contains a list of the commands executed. Once a command is selected or entered in history, clicking *Compile* starts the execution of the command. Use the F2 keyboard shortcut key to jump directly to the command line. In addition, there is a context menu associated with the command line (Figure 2.5):

The *Stop* button stops the current process session.

The next four buttons open the option setting, the smart slider, type setting, and the message setting dialog box.

The last button clears the content area (Output Window).

**Figure 2.5 Command line Context Menu**



## Status Bar

When pointing to a button in the toolbar or a menu entry, the message area displays the function of the button or menu entry being pointed to.

**Figure 2.6 Status Bar**



## Menu Bar

Table 2.1 lists and describes the menus available in the menu bar (Figure 2.7):

**Table 2.1 Menus in the Menu Bar**

Menu Entry	Description
File	Contains entries to manage application configuration files.
Compiler	Contains entries to set the application options.
View	Contains entries to customize the application window output.
Help	A standard Windows Help menu.

Figure 2.7 Menu Bar

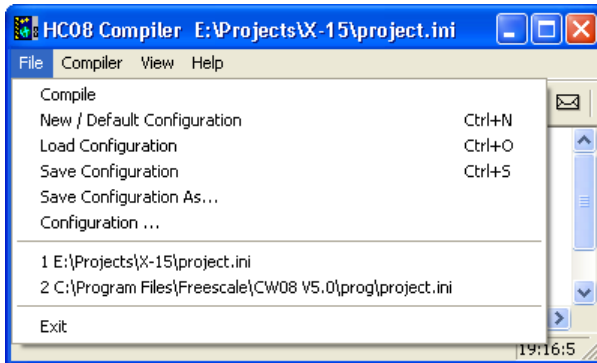


## File Menu

Save or load configuration files from the File Menu (Figure 2.8). A configuration file contains the following information:

- The application option settings specified in the application dialog boxes
- The Message Settings that specify which messages to display and which messages to treat as error messages
- The list of the last command line executed and the current command line being executed
- The window position
- The Tips of the Day settings, including if enabled at startup and which is the current entry

Figure 2.8 File Menu



Configuration files are text files which use the standard extension \*.ini. A developer can define as many configuration files as required for a project. The developer can also switch between the different configuration files using the *File > Load Configuration* and *File > Save Configuration* menu entries or the corresponding toolbar buttons.

## Editor Settings dialog box

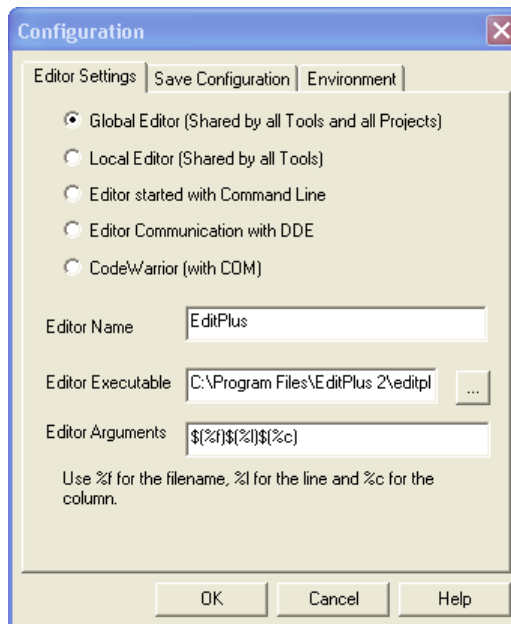
The Editor Settings dialog box has a main selection entry. Depending on the main type of editor selected, the content below changes.

These main Editor Setting entries are described on the following pages.

## Global Editor configuration

The *Global Editor* (Figure 2.9) is shared among all tools and projects on one work station. It is stored in the `mcutools.ini` global initialization file in the [Editor] section. Some Modifiers are specified in the editor command line.

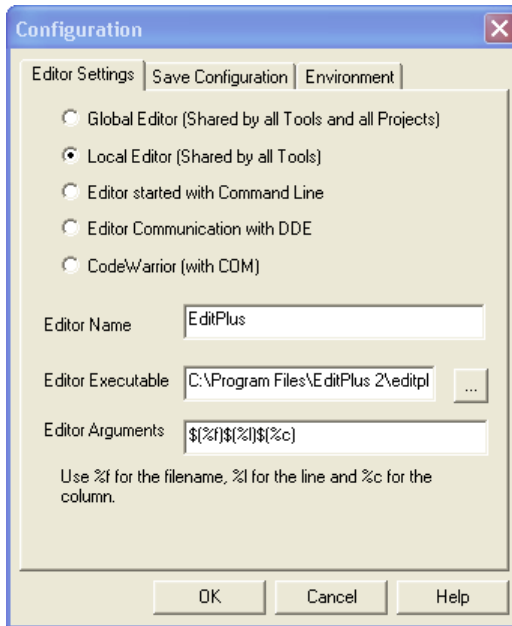
**Figure 2.9** Global Editor configuration



## Local Editor configuration

The Local Editor (Figure 2.10) is shared among all tools using the same project file. When an entry of the Global or Local configuration is stored, the behavior of the other tools using the same entry also changes when these tools are restarted.

**Figure 2.10 Local Editor configuration**



## **Editor started with Command Line**

When this editor type (Figure 2.11) is selected, a separate editor is associated with the application for error feedback. The configured editor is not used for error feedback.

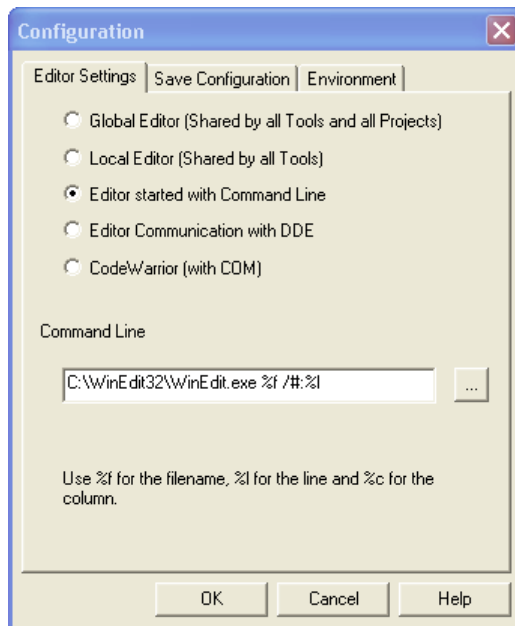
Enter the command that starts the editor.

The format of the editor command depends on the syntax. Some Modifiers are specified in the editor command line to refer to a line number in the file. (See the Modifiers section below.)

The format of the editor command depends upon the syntax that is used to start the editor.



Figure 2.11 Editor Started with Command Line



## Examples

(also look at the notes below)

For CodeWright V6 version use (with an adapted path to the cw32 . exe file):

```
C:\cw32.exe %f -g%l
```

For the WinEdit 32-bit version use (with an adapted path to the winedit.exe file):

```
C:\WinEdit32\WinEdit.exe %f /#:%l
```

## Editor Started with DDE

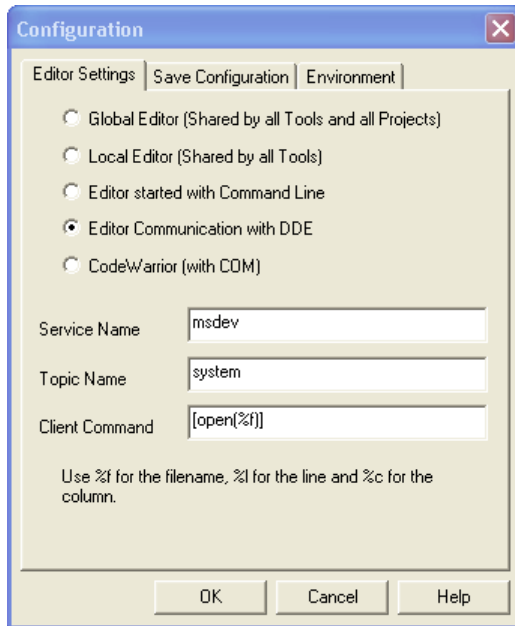
Enter the service and topic names and the client command for the DDE connection to the editor (Microsoft Develop Studio - Figure 2.12 or UltraEdit-32 - Figure 2.13). The entries for Topic Name and Client Command can have modifiers for the filename, line number, and column number as explained in Modifiers.

## Graphical User Interface

### Menu Bar

---

**Figure 2.12 Editor Started with DDE (Microsoft Developer Studio)**



For Microsoft Developer Studio, use the settings in Listing 2.5.

#### **Listing 2.5 .Microsoft Developer Studio configuration**

---

```
Service Name   : msdev
Topic Name     : system
Client Command : [open(%f)]
```

---

UltraEdit-32 is a powerful shareware editor. It is available from [www.idmcomp.com](http://www.idmcomp.com) or [www.ultraedit.com](http://www.ultraedit.com), email [idm@idmcomp.com](mailto:idm@idmcomp.com). For UltraEdit, use the following settings (Listing 2.6).

#### **Listing 2.6 UltraEdit-32 editor settings.**

---

```
Service Name   : UEDIT32
Topic Name     : system
Client Command : [open("%f/%l/%c")]
```

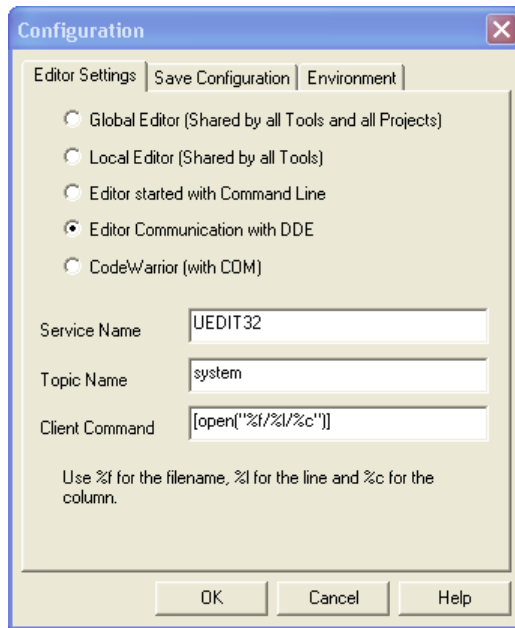
---

---

**NOTE** The DDE application (e.g., Microsoft Developer Studio or UltraEdit) has to be started or otherwise the DDE communication will fail.

---

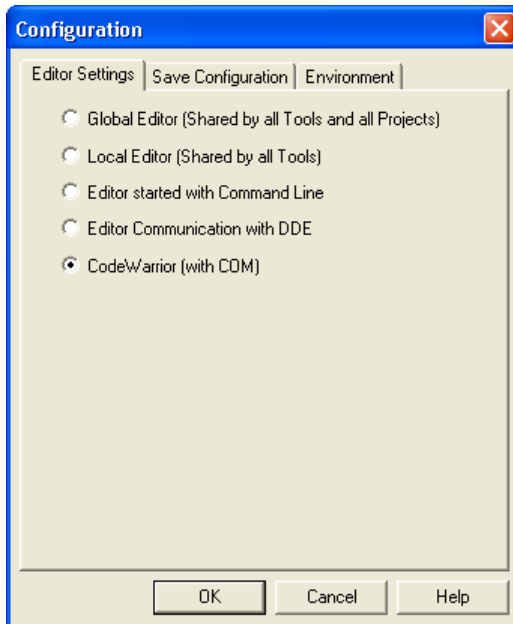
**Figure 2.13 Editor Started with DDE (UltraEdit-32)**



### CodeWarrior (with COM)

If CodeWarrior with COM (Figure 2.14) is enabled, the CodeWarrior IDE (registered as COM server by the installation script) is used as the editor.

Figure 2.14 CodeWarrior (with COM)



## Modifiers

The configuration must contain modifiers that instruct the editor which file to open and at which line.

- The %f modifier refers to the name of the file (including path) where the message has been detected.
- The %l modifier refers to the line number where the message has been detected.
- The %c modifier refers to the column number where the message has been detected.

---

**NOTE** The %l modifier can only be used with an editor which is started with a line number as a parameter. This is not the case for WinEdit version 3.1 or lower or for the Notepad. When working with such an editor, start it with the filename as a parameter and then select the menu entry 'Go to' to jump on the line where the message has been detected. *In that case the editor command looks like:*

```
C:\WINAPPS\WINEDIT\Winedit.EXE %f
```

*Please check the editor manual to define the command line which should be used to start the editor.*

---

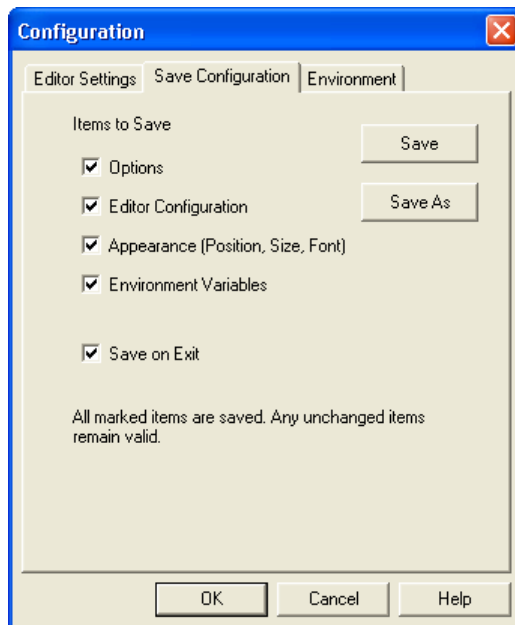
## Save Configuration dialog box

All save options are located on the second page of the configuration dialog box.

Use the Save Configuration dialog box to configure which parts of your configuration will be stored into a project file.

This Save Configuration dialog box (Figure 2.15) offers the following options:

**Figure 2.15 Save Configuration dialog box**



- Options  
The current option and message setting is saved when a configuration is written. By disabling this option, the last saved content remains valid.
- Editor Configuration  
The current editor setting is saved when a configuration is written. By disabling this option, the last saved content remains valid.
- Appearance  
This saves topics consisting of many parts such as the window position (only loaded at startup time) and the command line content and history. These settings are saved when a configuration is written.

## Graphical User Interface

### Menu Bar

---

- Environment Variables

The environment variable changes done in the Environment property sheet are saved.

---

**NOTE** By disabling selective options only some parts of a configuration file are written. For example, when the best options are found, the save option mark is removed. Subsequent future save commands will no longer modify the options.

---

- Save on Exit

The application writes the configuration on exit. No question dialog box appears to confirm this operation. If this option is not set, the application will not write the configuration at exit, even if options or another part of the configuration have changed. No confirmation appears in either case when closing the application.

---

**NOTE** Most settings are stored in the configuration file only.  
The only exceptions are:

- The recently used configuration list.
- All settings in this dialog box.

---

---

**NOTE** The application configurations can (and in fact are intended to) coexist in the same file as the project configuration of UltraEdit-32. When an editor is configured by the shell, the application reads this content out of the project file, if present. The project configuration file of the shell is named `project.ini`. This filename is also suggested (but not required) to be used by the application.

---

## Environment Configuration Dialog Box

This Environment Configuration dialog box (Figure 2.16) is used to configure the environment. The content of the dialog box is read from the actual project file out of the section [Environment Variables].

The following environment variables are available (Listing 2.7):

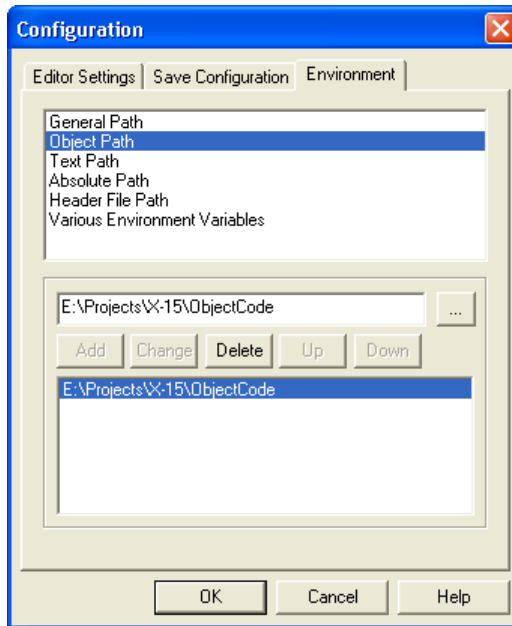
### Listing 2.7 Environment variables

---

```
General Path:      GENPATH
Object Path:       OBJPATH
Text Path:         TEXTPATH
Absolute Path:    ABSPATH
Header File Path: LIBPATH
Various Environment Variables: other variables not mentioned above.
```

---

Figure 2.16 Environment Configuration dialog box



The following buttons are available on this dialog box (Table 2.2):

Table 2.2 Functions of the buttons on the Environment Configuration dialog box

Button	Function
Add	Adds a new line or entry
Change	Changes a line or entry
Delete	Deletes a line or entry
Up	Moves a line or entry up
Down	Moves a line or entry down

The variables are written to the project file only if the *Save* button is pressed (or use *File > Save Configuration*, or *CTRL-S*). In addition, the environment is specified if it is to be written to the project in the *Save Configuration* dialog box.

## Compiler Menu

This menu (Figure 2.17) enables the application to be customized. Application options are graphically set as well as defining the optimization level. Table 2.3 defines the Compiler Menu options:

Figure 2.17 Compiler Menu

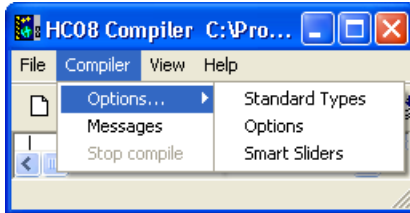


Table 2.3 Compiler Menu options

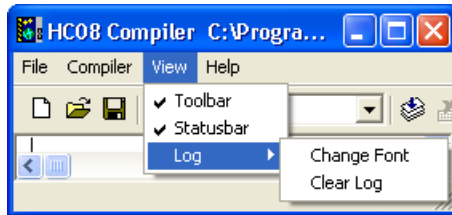
Menu entry	Description
Options...	Allows you to customize the application. You can graphically set or reset options. The next three entries are available when <i>Options...</i> is selected:
Standard Types	Allows you to specify the size you want to associate with each ANSI C standard type. (See "Standard Types dialog box" on page 98.)
Advanced	Allows you to define the options which must be activated when processing an input file. (See "Option Settings dialog box" on page 99.)
Smart Slider	Allows you to define the optimization level you want to reach when processing the input file. (See "Compiler Smart Control dialog box" on page 101.)
Messages	Opens a dialog box, where the different error, warning, or information messages are mapped to another message class. (See "Message Settings dialog box" on page 103.)
Stop Compilation	Immediately stops the current processing session.



## View Menu

The View Menu (Figure 2.18) enables you to customize the application window. You can define things such as displaying or hiding the status or toolbar. You can also define the font used in the window, or clear the window. Table 2.4 lists the View Menu options.

**Figure 2.18 View Menu**



**Table 2.4 View Menu options**

Menu entry	Description
Toolbar	Switches display from the toolbar in the application window.
Status Bar	Switches display from the status bar in the application window.
Log...	Allows you to customize the output in the application window content area. The following entries are available when <i>Log...</i> is selected:
Change Font	Opens a standard font selection box. The options selected in the font dialog box are applied to the application window content area.
Clear Log	Allows you to clear the application window content area.

## Help Menu

The Help Menu (Figure 2.19) enables you to either display or not display the Tip of the Day dialog box application startup. In addition, it provides a standard Windows Help entry and an entry to an About box. Table 2.5 defines the Help Menu options:

## Graphical User Interface

### Standard Types dialog box

Figure 2.19 Help Menu

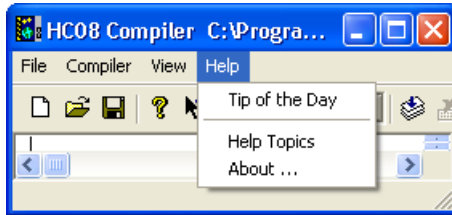


Table 2.5 Help Menu Options

Menu entry	Description
Tip of the Day	Switches on or off the display of a Tip of the Day during startup.
Help Topics	Standard Help topics.
About...	Displays an About box with some version and license information.

## Standard Types dialog box

The Standard Types dialog box (Figure 2.20) enables you to define the size you want to associate to each ANSI C standard type. You can also use the -T: Flexible Type Management compiler option to configure ANSI-C standard type sizes.

---

**NOTE** Not all formats may be available for a target. In addition, there has to be at least one type for each size. For example, it is illegal to specify all types to a size of 32 bits. There is no type for 8 bits and 16 bits available for the Compiler. Note that if the HIWARE object-file Format is used instead of the ELF/DWARF object-file Format, the HIWARE Format does not support a size greater than 1 for the char type.

---

The following rules (Listing 2.8) apply when you modify the size associated with an ANSI-C standard type:

Listing 2.8 Size relationships for the ANSI-C standard types.

---

```
sizeof(char)  <= sizeof(short)
sizeof(short) <= sizeof(int)
sizeof(int)   <= sizeof(long)
sizeof(long)  <= sizeof(long long)
```

---

```
sizeof(float) <= sizeof(double)  
sizeof(double) <= sizeof(long double)
```

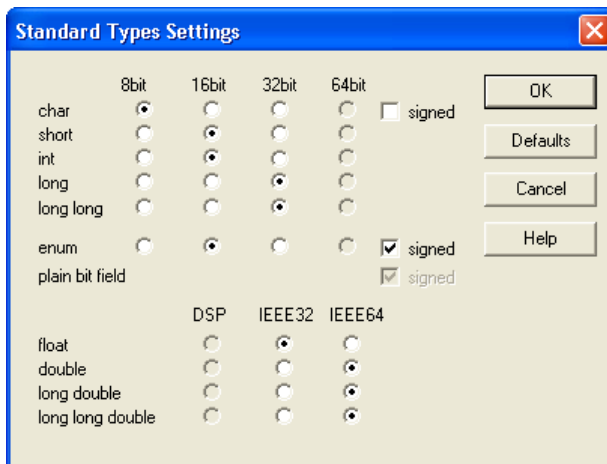
Enumerations must be smaller than or equal to 'int'.

The *signed* check box enables you to specify whether the `char` type must be considered as signed or unsigned for your application.

The *Default* button resets the size of the ANSI C standard types to their default values.

The ANSI C standard type default values depend on the target processor.

Figure 2.20 Standard Types Dialog Box



## Option Settings dialog box

The Option Settings dialog box (Figure 2.21) enables you to set or reset application options. The possible command line option is also displayed in the lower display area. The available options are arranged into different groups. A sheet is available for each of these groups. The content of the list box depends on the selected sheet (not all groups may be available). Table 2.6 lists the Option Settings dialog box selections.

## Graphical User Interface

### Option Settings dialog box

Figure 2.21 Option Settings dialog box

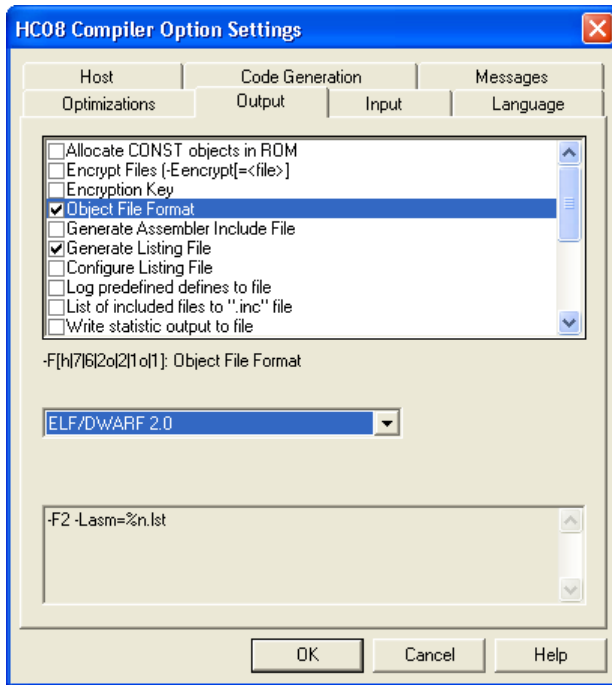


Table 2.6 Option Settings dialog box selections

Group	Description
Optimizations	Lists optimization options.
Output	Lists options related to the output files generation (which kind of file should be generated).
Input	Lists options related to the input file.
Language	Lists options related to the programming language (ANSI-C)
Target	Lists options related to the target processor.
Host	Lists options related to the host operating system.
Code Generation	Lists options related to code generation (memory models, float format, ...).

**Table 2.6 Option Settings dialog box selections (continued)**

<b>Group</b>	<b>Description</b>
Messages	Lists options controlling the generation of error messages.
Various	Lists options not related to the above options.

An application option is set when its check box is checked. To obtain a more detailed explanation about a specific option, select the option and press the F1 key or the help button. To select an option, click once on the option text. The option text is then displayed color-inverted. When the dialog box is opened and no option is selected, pressing the F1 key or the help button shows the help for this dialog box.

---

**NOTE** When options requiring additional parameters are selected, you can open an edit box or an additional sub window where the additional parameter is set. For example for the option ‘Write statistic output to file...’, available in the Output sheet.

---

## Compiler Smart Control dialog box

The Compiler Smart Control Dialog Box (Figure 2.22) enables you to define the optimization level you want to reach during compilation of the input file. Five sliders are available to define the optimization level. See Table 2.7.

## Graphical User Interface

Compiler Smart Control dialog box

Figure 2.22 Compiler Smart Control dialog box

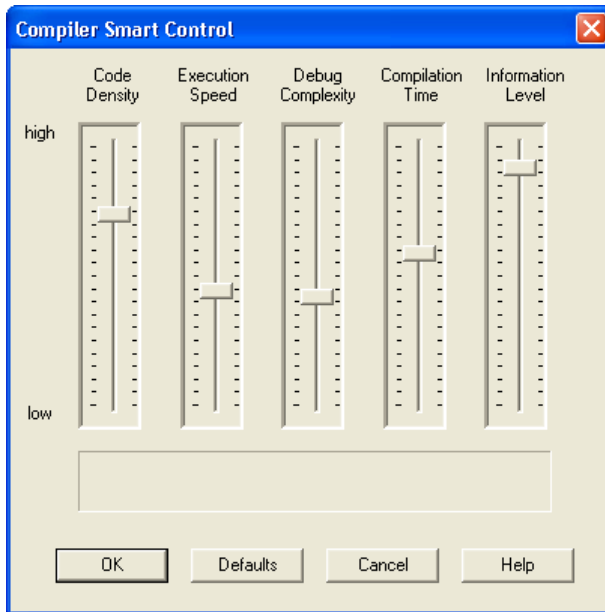


Table 2.7 Compiler Smart Control dialog box controls

Slider	Description
Code Density	Displays the code density level expected. A high value indicates highest code efficiency (smallest code size).
Execution Speed	Displays the execution speed level expected. A high value indicates fastest execution of the code generated.
Debug Complexity	Displays the debug complexity level expected. A high value indicates complex debugging. For example, assembly code corresponds directly to the high-level language code.
Compilation Time	Displays the compilation time level expected. A higher value indicates longer compilation time to produce the object file, e.g., due to high optimization.
Information Level	Displays the level of information messages which are displayed during a Compiler session. A high value indicates a verbose behavior of the Compiler. For example, it will inform with warnings and information messages.

There is a direct link between the first four sliders in this window. When you move one slider, the positions of the other three are updated according to the modification.

The command line is automatically updated with the options set in accordance with the settings of the different sliders.

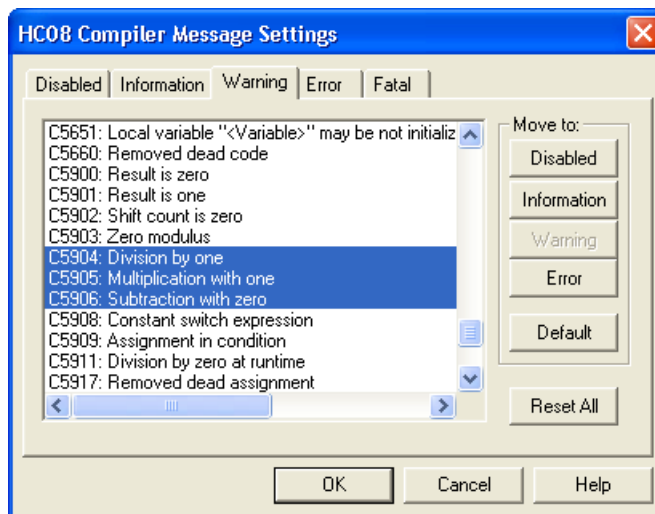
## Message Settings dialog box

The Message Settings dialog box (Figure 2.23) enables you to map messages to a different message class.

Some buttons in the dialog box may be disabled. (For example, if an option cannot be moved to an Information message, the 'Move to: Information' button is disabled.)

Table 2.8 lists and describes the buttons available in this dialog box.

Figure 2.23 Message Settings dialog box



**Table 2.8 Message Settings dialog box buttons**

<b>Button</b>	<b>Description</b>
Move to: Disabled	The messages selected will be disabled. The message will not occur any longer.
Move to: Information	The messages selected will be changed to information messages.
Move to: Warning	The messages selected will be changed to warning messages.
Move to: Error	The messages selected will be changed to error messages.
Move to: Default	The messages selected will be changed to their default message kind.
Reset All	Resets all messages to their default message kind.
OK	Exits this dialog box and accepts the changes made.
Cancel	Exits this dialog box without accepting the changes made.
Help	Displays online help about this dialog box.

A panel is available for each error message class. The content of the list box depends on the selected panel. Table 2.9 lists the definitions for the message groups.

**Table 2.9 Message Group definitions**

<b>Message group</b>	<b>Description</b>
Disabled	Lists all disabled messages. That means messages displayed in the list box will not be displayed by the application.
Information	Lists all information messages. Information messages inform about action taken by the application.
Warning	Lists all warning messages. When a warning message is generated, processing of the input file continues.



Table 2.9 Message Group definitions (*continued*)

Message group	Description
Error	Lists all error messages. When an error message is generated, processing of the input file continues.
Fatal	Lists all fatal error messages. When a fatal error message is generated, processing of the input file stops immediately. Fatal messages cannot be changed and are only listed to call context help.

Each message has its own prefix (e.g., ‘C’ for Compiler messages, ‘A’ for Assembler messages, ‘L’ for Linker messages, ‘M’ for Maker messages, ‘LM’ for Libmaker messages) followed by a 4- or 5-digit number. This number allows an easy search for the message both in the manual or on-line help.

## Changing the Class associated with a Message

You can configure your own mapping of messages in the different classes. For that purpose you can use one of the buttons located on the right hand of the dialog box. Each button refers to a message class. To change the class associated with a message, you have to select the message in the list box and then click the button associated with the class where you want to move the message.

### Example

---

#### To define a warning message as an error message:

1. Click the *Warning* panel to display the list of all warning messages in the list box.
2. Click on the message you want to change in the list box to select the message.
3. Click *Error* to define this message as an error message.

---

**NOTE** Messages cannot be moved to or from the fatal error class.

---

---

**NOTE** The ‘*Move to*’ buttons are active only when messages that can be moved are selected. When one message is marked which cannot be moved to a specific group, the corresponding ‘*Move to*’ button is disabled (grayed).

---

## Graphical User Interface

### About... dialog box

---

If you want to validate the modification you have performed in the error message mapping, close the 'Message Settings' dialog box using the 'OK' button. If you close it using the 'Cancel' button, the previous message mapping remains valid.

## Retrieving Information about an Error Message

You can access information about each message displayed in the list box. Select the message in the list box and then click *Help* or the *F1* key. An information box is opened. The information box contains a more detailed description of the error message, as well as a small example of code that may have generated the error message. If several messages are selected, a help for the first is shown. When no message is selected, pressing the *F1* key or the help button shows the help for this dialog box.

## About... dialog box

The About... dialog box is opened by selecting *Help->About...* The About box contains information regarding your application. The current directory and the versions of subparts of the application are also shown. The main version is displayed separately on top of the dialog box.

Use the 'Extended Information' button to get license information about all software components in the same directory as that of the executable file.

Click OK to close this dialog box.

---

**NOTE** During processing, the sub-versions of the application parts cannot be requested. They are only displayed if the application is inactive.

---

## Specifying the Input File

There are different ways to specify the input file. During the compilation, the options will be set according to the configuration established in the different dialog boxes.

Before starting to compile a file make sure you have associated a working directory with your editor.

## Use the Command Line in the Toolbar to Compile

The command line can be used to compile a new file and to open a file that has already been compiled.

### Compiling a new file

A new filename and additional Compiler options are entered in the command line. The specified file will be compiled as soon as the *Compile* button in the toolbar is selected or the Enter key is pressed.

### Compiling a file which has already been compiled

The previously executed command is displayed using the arrow on the right side of the command line. A command is selected by clicking on it. It appears in the command line. The specified file is compiled as soon as the *Compile* button in the toolbar is clicked.

### Use the Entry File -> Compile

When the menu entry *File | Compile* is selected, a standard open file box is displayed. Use this to locate the file you want to compile. The selected file is compiled as soon as the standard open file box is closed using the *Open* button.

### Use Drag and Drop

A filename is dragged from an external application (for example the File Manager/ Explorer) and dropped into the Compiler window. The dropped file is compiled as soon as the mouse button is released in the Compiler window. If a file being dragged has the “\*.ini” extension, it is considered to be a configuration and it is immediately loaded and not compiled. To compile a C file with the “\*.ini” extension, use one of the other methods to compile it.

## Message/Error Feedback

There are several ways to check where different errors or warnings have been detected after compilation. Listing 2.9 lists the format of the error messages and Listing 2.10 is a typical example of an error message.

## Graphical User Interface

### Specifying the Input File

---

#### Listing 2.9 Format of an error message

---

```
>> <FileName>, line <line number>, col <column number>, pos <absolute  
  position in file>  
<Portion of code generating the problem>  
<message class><message number>: <Message string>
```

---

#### Listing 2.10 Example of an error message

---

```
>> in "C:\DEMO\fibo.c", line 30, col 10, pos 428  
  EnableInterrupts  
  WHILE (TRUE) {  
    (  
INFORMATION C4000: Condition always TRUE
```

---

See also the `-WmsgFi` (`-WmsgFiv`, `-WmsgFim`): Set Message Format for Interactive Mode and `-WmsgFb` (`-WmsgFbi`, `-WmsgFbm`): Set Message File Format for Batch Mode compiler options for different message formats.

## Use Information from the Compiler Window

Once a file has been compiled, the Compiler window content area displays the list of all the errors or warnings that were detected.

Use your usual editor to open the source file and correct the errors.

## Use a User-Defined Editor

You must first configure the editor you want to use for message/error feedback in the *Configuration* dialog box before you begin the compile process. Once a file has been compiled, double-click on an error message. The selected editor is automatically activated and points to the line containing the error.

# Environment

---

This Chapter describes all the environment variables. Some environment variables are also used by other tools (e.g., Linker or Assembler). Consult the respective manual for more information.

The major sections in this chapter are:

- Current Directory
- Environment Macros
- Global Initialization File (mcutools.ini)
- Local Configuration File (usually project.ini)
- Paths
- Line Continuation
- Environment Variable Details

Parameters are set in an environment using environment variables. There are three ways to specify your environment:

1. The current project file with the [Environment Variables] section. This file may be specified on Tool startup using the `-Prod: Specify Project File at Startup` option.
2. An optional 'default.env' file in the current directory. This file is supported for backwards compatibility. The filename is specified using the `ENVIRONMENT: Environment File Specification` variable. Using the default.env file is not recommended.
3. Setting environment variables on system level (DOS level). This is not recommended.

The syntax for setting an environment variable is (Listing 3.1):

Parameter: `<KeyName>=<ParamDef>` (no spaces)

---

**NOTE** *Normally no white space is allowed in the definition of an environment variable.*

---

## Listing 3.1 Setting the GENPATH environment variable

---

```
GENPATH=C:\INSTALL\LIB;D:\PROJECTS\TESTS;/usr/local/lib;  
/home/me/my_project
```

---

## Environment

### Current Directory

---

Parameters may be defined in several ways:

- Using system environment variables supported by your operating system.
- Putting the definitions into the actual project file in the section named [Environment Variables].
- Putting the definitions in a file named `default.env` in the default directory.

---

**NOTE** The maximum length of environment variable entries in the `default.env` file is 4096 characters.

---

- Putting the definitions in a file given by the value of the `ENVIRONMENT` system environment variable.

---

**NOTE** The default directory mentioned above is set using the `DEFAULTDIR`: Default Current Directory system environment variable.

---

When looking for an environment variable, all programs first search the system environment, then the `default.env` file, and finally the global environment file defined by `ENVIRONMENT`. If no definition is found, a default value is assumed.

---

**NOTE** The environment may also be changed using the `-Env`: Set Environment Variable option.

---

---

**NOTE** Make sure that there are no spaces at the end of any environment variable declaration.

---

## Current Directory

The most important environment for all tools is the current directory. The current directory is the base search directory where the tool starts to search for files (e.g., for the `default.env` file).

The current directory of a tool is determined by the operating system or by the program which launches another one.

- For the UNIX operating system, the current directory of an launched executable is also the current directory from where the binary file has been started.
- For MS Windows based operating systems, the current directory definition is defined as follows:
  - If the tool is launched using the File Manager or Explorer, the current directory is the location of the launched executable.

- If the tool is launched using an Icon on the Desktop, the current directory is the one specified and associated with the Icon.
- If the tool is launched by another launching tool with its own current directory specification (e.g., an editor), the current directory is the one specified by the launching tool (e.g., current directory definition).
- For the tools, the current directory is where the local project file is located. Changing the current project file also changes the current directory if the other project file is in a different directory. Note that browsing for a C file does not change the current directory.

To overwrite this behavior, use the DEFAULTDIR: Default Current Directory environment variable.

The current directory is displayed, with other information, using the “-V: Prints the Compiler Version” compiler option and in the About... dialog box.

## Environment Macros

You can use macros in your environment settings (Listing 3.2).

### Listing 3.2 Using Macros for setting environment variables

---

```
MyVAR=C:\test
TEXTPATH=${MyVAR}\txt
OBJPATH=${MyVAR}\obj
```

---

In the example above, TEXTPATH is expanded to 'C:\test\txt' and OBJPATH is expanded to 'C:\test\obj'. You can use \$( ) or \${ }. However, the referenced variable must be defined.

Special variables are also allowed (special variables are always surrounded by { } and they are case-sensitive). In addition, the variable content contains the directory separator '\'. The special variables are:

- {Compiler}

That is the path of the executable one directory level up if the executable is 'C:\Freescale\prog\linker.exe', and the variable is 'C:\Freescale\'.

- {Project}

Path of the current project file. This is used if the current project file is 'C:\demo\project.ini', and the variable contains 'C:\demo\'.

- {System}

## Environment

### Global Initialization File (*mcutools.ini*)

---

This is the path where your Windows system is installed, e.g., 'C:\WINNT\ '.

## Global Initialization File (*mcutools.ini*)

All tools store some global data into the file *mcutools.ini*. The tool first searches for the *mcutools.ini* file in the directory of the tool itself (path of the executable). If there is no *mcutools.ini* file in this directory, the tool looks for an *mcutools.ini* file in the MS Windows installation directory (e.g., C:\WINDOWS).

### Listing 3.3 Typical Global Initialization File Locations

---

```
C:\WINDOWS\mcutools.ini  
D:\INSTALL\prog\mcutools.ini
```

---

If a tool is started in the D:\INSTALL\prog directory, the project file that is used is located in the same directory as the tool (D:\INSTALL\prog\mcutools.ini).

If the tool is started outside the D:\INSTALL\prog directory, the project file in the Windows directory is used (C:\WINDOWS\mcutools.ini).

Global Configuration-File Entries documents the sections and entries you can include in the *mcutools.ini* file.

## Local Configuration File (usually *project.ini*)

All the configuration properties are stored in the configuration file. The same configuration file is used by different applications.

The shell uses the configuration file with the name "project.ini" in the current directory only. When the shell uses the same file as the compiler, the Editor Configuration is written and maintained by the shell and is used by the compiler. Apart from this, the compiler can use any filename for the project file. The configuration file has the same format as the windows \*.ini files. The compiler stores its own entries with the same section name as those in the global *mcutools.ini* file. The compiler backend is encoded into the section name, so that a different compiler backend can use the same file without any overlapping. Different versions of the same compiler use the same entries. This plays a role when options, only available in one version, must be stored in the configuration file. In such situations, two files must be maintained for each different compiler version. If no incompatible options are enabled when the file is last saved, the same file may be used for both compiler versions.

The current directory is always the directory where the configuration file is located. If a configuration file in a different directory is loaded, the current directory also changes.



When the current directory changes, the entire `default.env` file is reloaded. When a configuration file is loaded or stored, the options in the environment variable `COMPOPTIONS` are reloaded and added to the project options. This behavior is noticed when different `default.env` files exist in different directories, each containing incompatible options in the `COMPOPTIONS` variable.

When a project is loaded using the first `default.env`, its `COMPOPTIONS` are added to the configuration file. If this configuration is stored in a different directory where a `default.env` exists with incompatible options, the compiler adds options and remarks the inconsistency. You can remove the option from the configuration file with the option settings dialog box. You can also remove the option from the `default.env` with the shell or a text editor, depending which options will be used in the future.

At startup, there are two ways to load a configuration:

- Use the `-Prod: Specify Project File at Startup` command line option
- The `project.ini` file in the current directory.

If the `-Prod` option is used, the current directory is the directory the project file is in. If the `-Prod` option is used with a directory, the `project.ini` file in this directory is loaded.

Local Configuration-File Entries documents the sections and entries you can include in a `project.ini` file.

## Paths

A path list is a list of directory names separated by semicolons. Path names are declared using the following EBNF syntax (Listing 3.4). Most environment variables contain path lists directing where to look for files (Listing 3.5).

### Listing 3.4 EBNF path syntax

---

```
PathList = DirSpec { ; DirSpec } .
DirSpec  = [*] DirectoryName .
```

---

### Listing 3.5 Environment variable path list with four possible paths.

---

```
GENPATH=C:\INSTALL\LIB;D:\PROJECTS\TESTS;/usr/local/lib;
/home/me/my_project
```

---

If a directory name is preceded by an asterisk ( `"*"` ), the program recursively searches that entire directory tree for a file, not just the given directory itself. The directories are searched in the order they appear in the path list.

## Environment

### Line Continuation

---

#### Listing 3.6 Setting an environment variable using recursive searching

---

```
LIBPATH=*C:\INSTALL\LIB
```

---

**NOTE** Some DOS environment variables (like GENPATH, LIBPATH, etc.) are used.

If you work with CodeWright, you can set the environment using a <project>.pjf file in your project directory. This enables you to have different projects in different directories, each with its own environment.

**NOTE** When using WinEdit, do *not* set the system environment variable DEFAULTDIR: Default Current Directory. If you do so, and this variable does not contain the project directory given in WinEdit's project configuration, files might not be placed where you expect them to be.

---

## Line Continuation

It is possible to specify an environment variable in an environment file (default.env) over different lines using the line continuation character '\ ' (see Listing 3.7).

#### Listing 3.7 Specifying an environment variable using line continuation characters

---

```
OPTIONS=\  
  -W2 \  
  -Wpd
```

---

This is the same as:

```
OPTIONS=-W2 -Wpd
```

But this feature may not work well using it together with paths, e.g.:

```
GENPATH=.\ \  
TEXTFILE=.\txt
```

will result in:

```
GENPATH=. TEXTFILE=.\txt
```

To avoid such problems, use a semicolon ' ; ' at the end of a path if there is a ' \ ' at the end (Listing 3.8):

**Listing 3.8 Using a semicolon to allow a multiline environment variable**

---

```
GENPATH= . \ ;  
TEXTFILE= . \txt
```

---

## Environment Variable Details

The remainder of this chapter describes each of the possible environment variables. Table 3.1 lists these description topics in their order of appearance for each environment variable.

**Table 3.1 Environment Variables—documentation topics**

Topic	Description
Tools	Lists tools that use this variable.
Synonym	A synonym exists for some environment variables. Those synonyms may be used for older releases of the Compiler and will be removed in the future. A synonym has lower precedence than the environment variable.
Syntax	Specifies the syntax of the option in an EBNF format.
Arguments	Describes and lists optional and required arguments for the variable.
Default	Shows the default setting for the variable or none.
Description	Provides a detailed description of the option and how to use it.
Example	Gives an example of usage, and the effects of the variable where possible. The example shows an entry in the default.env for a PC.
See also	Names related sections.

## Environment

### Environment Variable Details

---

## COMPOPTIONS: Default Compiler Options

### Tools

Compiler

### Synonym

HICOMPOPTIONS

### Syntax

COMPOPTIONS={<option>}

### Arguments

<option>: Compiler command-line option

### Default

None

### Description

If this environment variable is set, the Compiler appends its contents to its command line each time a file is compiled. It is used to globally specify options that should always be set. This frees you from having to specify them for every compilation.

---

**NOTE** It is not recommended to use this environment variable if the Compiler used is version 5.x, because the Compiler adds the options specified in the COMPOPTIONS variable to the options stored in the `project.ini` file.

---

### Listing 3.9 Setting default values for environment variables (not recommended)

---

```
COMPOPTIONS=-w2 -wpd
```

---

### See also

Compiler Options

## **COPYRIGHT: Copyright entry in object file**

### **Tools**

Compiler, Assembler, Linker, or Librarian

### **Synonym**

None

### **Syntax**

`COPYRIGHT=<copyright>`

### **Arguments**

`<copyright>`: copyright entry

### **Default**

None

### **Description**

Each object file contains an entry for a copyright string. This information is retrieved from the object files using the decoder.

### **Example**

```
COPYRIGHT=Copyright by Freescale
```

### **See also**

#### **environmental variables:**

- USERNAME: User Name in Object File
- INCLUDETIME: Creation Time in Object File

## Environment

### Environment Variable Details

---

## DEFAULTDIR: Default Current Directory

### Tools

Compiler, Assembler, Linker, Decoder, Debugger, Librarian, Maker, or Burner

### Synonym

None

### Syntax

DEFAULTDIR=<directory>

### Arguments

<directory>: Directory to be the default current directory

### Default

None

### Description

Specifies the default directory for all tools. All the tools indicated above will take the specified directory as their current directory instead of the one defined by the operating system or launching tool (e.g., editor).

---

**NOTE** This is an environment variable on a system level (global environment variable). It cannot be specified in a default environment file (`default.env`).

---

Specifying the default directory for all tools in the CodeWarrior suite:

```
DEFAULTDIR=C:\INSTALL\PROJECT
```

### See also

Current Directory

Global Initialization File (`mcutools.ini`)

## **ENVIRONMENT: Environment File Specification**

### **Tools**

Compiler, Linker, Decoder, Debugger, Librarian, Maker, or Burner

### **Synonym**

HIENVIRONMENT

### **Syntax**

ENVIRONMENT=<file>

### **Arguments**

<file>: filename with path specification, without spaces

### **Default**

None

### **Description**

This variable is specified on a system level. The application looks in the current directory for an environment file named `default.env`. Using `ENVIRONMENT` (e.g., set in the `autoexec.bat` (DOS) or `*.cshrc` (UNIX)), a different filename may be specified.

---

**NOTE** This is an environment variable on a system level (global environment variable). It cannot be specified in a default environment file (`default.env`).

---

### **Example**

ENVIRONMENT=\Freescale\prog\global.env

## Environment

### Environment Variable Details

---

## ERRORFILE: Error filename Specification

### Tools

Compiler, Assembler, Linker, or Burner

### Synonym

None

### Syntax

```
ERRORFILE=<filename>
```

### Arguments

<filename>: filename with possible format specifiers

### Description

The `ERRORFILE` environment variable specifies the name for the error file.

Possible format specifiers are:

`%n`: Substitute with the filename, without the path.

`%p`: Substitute with the path of the source file.

`%f`: Substitute with the full filename, i.e., with the path and name (the same as `%p%n`).

A notification box is shown in the event of an improper error filename.

### Examples

```
ERRORFILE=MyErrors.err
```

Lists all errors into the `MyErrors.err` file in the current directory.

```
ERRORFILE=\tmp\errors
```

Lists all errors into the `errors` file in the `\tmp` directory.

```
ERRORFILE=%f.err
```

Lists all errors into a file with the same name as the source file, but with the `*.err` extension, into the same directory as the source file. If you compile a file



such as `sources\test.c`, an error list file, `\sources\test.err`, is generated.

```
ERRORFILE=\dir1\%n.err
```

For a source file such as `test.c`, an error list file with the name `\dir1\test.err` is generated.

```
ERRORFILE=%p\errors.txt
```

For a source file such as `\dir1\dir2\test.c`, an error list file with the name `\dir1\dir2\errors.txt` is generated.

If the `ERRORFILE` environment variable is not set, the errors are written to the `EDOUT` file in the current directory.

## Environment

### Environment Variable Details

---

## GENPATH: #include “File” Path

### Tools

Compiler, Linker, Decoder, Debugger, or Burner

### Synonym

HIPATH

### Syntax

GENPATH={ <path> }

### Arguments

<path>: Paths separated by semicolons, without spaces

### Default

Current directory

### Description

If a header file is included with double quotes, the Compiler searches first in the current directory, then in the directories listed by GENPATH, and finally in the directories listed by LIBRARYPATH

---

**NOTE** If a directory specification in this environment variable starts with an asterisk (" \* "), the whole directory tree is searched recursively depth first, i.e., all subdirectories and *their* subdirectories and so on are searched. Search order of the subdirectories is indeterminate within one level in the tree.

---

### Example

```
GENPATH=\sources\include;..\..\headers;\usr\local\lib
```

### See also

LIBRARYPATH: ‘include <File>’ Path environment variable

## **INCLUDETIME: Creation Time in Object File**

### **Tools**

Compiler, Assembler, Linker, or Librarian

### **Synonym**

None

### **Syntax**

INCLUDETIME= (ON | OFF)

### **Arguments**

ON: Include time information into object file

OFF: Do not include time information into object file

### **Default**

ON

### **Description**

Each object file contains a time stamp indicating the creation time and data as strings. Whenever a new file is created by one of the tools, the new file gets a new time stamp entry.

This behavior may be undesired if (for Software Quality Assurance reasons) a binary file compare has to be performed. Even if the information in two object files is the same, the files do not match exactly as the time stamps are not identical. To avoid such problems, set this variable to OFF. In this case, the time stamp strings in the object file for date and time are “none” in the object file.

The time stamp is retrieved from the object files using the decoder.

### **Example**

```
INCLUDETIME=OFF
```

### **See also**

#### **environment variables:**

- COPYRIGHT: Copyright entry in object file
- USERNAME: User Name in Object File

## Environment

### Environment Variable Details

---

## LIBRARYPATH: ‘include <File>’ Path

### Tools

Compiler, ELF tools (Burner, Linker, or Decoder)

### Synonym

LIBPATH

### Syntax

```
LIBRARYPATH={<path>}
```

### Arguments

<path>: Paths separated by semicolons, without spaces

### Default

Current directory

### Description

If a header file is included with double quotes, the Compiler searches first in the current directory, then in the directories given by GENPATH: #include “File” Path and finally in the directories given by LIBRARYPATH: ‘include <File>’ Path.

---

**NOTE** If a directory specification in this environment variable starts with an asterisk (“\*”), the whole directory tree is searched recursively depth first, i.e., all subdirectories and *their* subdirectories and so on are searched. Search order of the subdirectories is indeterminate within one level in the tree.

---

### Example

```
LIBRARYPATH=\sources\include;..\headers;\usr\local\lib
```

### See also

GENPATH: #include “File” Path environment variable

USELIBPATH: Using LIBPATH Environment Variable

Input Files

## **OBJPATH: Object File Path**

### **Tools**

Compiler, Linker, Decoder, Debugger, or Burner

### **Synonym**

None

### **Syntax**

`OBJPATH=<path>`

### **Default**

Current directory

### **Arguments**

`<path>`: Path without spaces

### **Description**

If the Compiler generates an object file, the object file is placed into the directory specified by `OBJPATH`. If this environment variable is empty or does not exist, the object file is stored into the path where the source has been found.

If the Compiler tries to generate an object file specified in the path specified by this environment variable but fails (e.g., because the file is locked), the Compiler will issue an error message.

If a tool (e.g., the Linker) looks for an object file, it first checks for an object file specified by this environment variable, then in `GENPATH: #include "File" Path`, and finally in `HIPATH`.

### **Example**

```
OBJPATH=\sources\obj
```

### **See also**

Output Files

## Environment

### Environment Variable Details

---

## TEXTPATH: Text File Path

### Tools

Compiler, Linker, or Decoder

### Synonym

None

### Syntax

TEXTPATH=<path>

### Arguments

<path>: Path without spaces

### Default

Current directory

### Description

If the Compiler generates a textual file, the file is placed into the directory specified by TEXTPATH. If this environment variable is empty or does not exist, the text file is stored into the current directory.

### Example

```
TEXTPATH=\sources\txt
```

### See also

Output Files

#### compiler options:

- -Li: List of Included Files
- -Lm: List of Included Files in Make Format
- -Lo: Object File List

## **TMP: Temporary Directory**

### **Tools**

Compiler, Assembler, Linker, Debugger, or Librarian

### **Synonym**

None

### **Syntax**

TMP=<directory>

### **Arguments**

<directory>: Directory to be used for temporary files

### **Default**

None

### **Description**

If a temporary file must be created, the ANSI function, `tmpnam()`, is used. This library function stores the temporary files created in the directory specified by this environment variable. If the variable is empty or does not exist, the current directory is used. Check this variable if you get the error message “Cannot create temporary file”.

---

**NOTE** This is an environment variable on a system level (global environment variable). It cannot be specified in a default environment file (`default.env`).

---

### **Example**

```
TMP=C:\TEMP
```

### **See also**

Current Directory

## Environment

### Environment Variable Details

---

## USELIBPATH: Using LIBPATH Environment Variable

### Tools

Compiler, Linker, or Debugger

### Synonym

None

### Syntax

USELIBPATH= (OFF | ON | NO | YES)

### Arguments

ON, YES: The environment variable LIBRARYPATH is used by the Compiler to look for system header files <\*.h>.

NO, OFF: The environment variable LIBRARYPATH is not used by the Compiler.

### Default

ON

### Description

This environment variable allows a flexible usage of the LIBRARYPATH environment variable as the LIBRARYPATH variable might be used by other software (e.g., version management PVCS).

### Example

```
USELIBPATH=ON
```

### See also

LIBRARYPATH: 'include <File>' Path environment variable



## **USERNAME: User Name in Object File**

### **Tools**

Compiler, Assembler, Linker, or, Librarian

### **Synonym**

None

### **Syntax**

USERNAME=<user>

### **Arguments**

<user>: Name of user

### **Default**

None

### **Description**

Each object file contains an entry identifying the user who created the object file. This information is retrievable from the object files using the decoder.

### **Example**

```
USERNAME=The Master
```

### **See also**

**environment variables:**

- COPYRIGHT: Copyright entry in object file
- INCLUDETIME: Creation Time in Object File

## **Environment**

### *Environment Variable Details*

---

# Files

---

This chapter describes input and output files and file processing.

“Input Files” on page 131

“Output Files” on page 132

“File Processing” on page 133

## Input Files

The following input files are described in this section:

- Source Files
- Include Files

### Source Files

The frontend takes any file as input. It does not require the filename to have a special extension. However, it is suggested that all your source filenames have the `*.c` extension and that all header files use the `*.h` extension. Source files are searched first in the Current Directory and then in the GENPATH: #include “File” Path directory.

### Include Files

The search for include files is governed by two environment variables: GENPATH: #include “File” Path and LIBRARYPATH: ‘include <File>’ Path. Include files that are included using double quotes as in:

```
#include "test.h"
```

are searched first in the current directory, then in the directory specified by the -I: Include File Path option, then in the directories given in the GENPATH: #include “File” Path environment variable, and finally in those listed in the LIBPATH or LIBRARYPATH: ‘include <File>’ Path environment variable. The current directory is set using the IDE, the Program Manager, or the DEFAULTDIR: Default Current Directory environment variable.

Include files that are included using angular brackets as in:

```
#include <stdio.h>
```

## Files

### Output Files

---

are searched for first in the current directory, then in the directory specified by the `-I` option, and then in the directories given in `LIBPATH` or `LIBRARYPATH`. The current directory is set using the IDE, the Program Manager, or the `DEFAULTDIR` environment variable.

## Output Files

The following output files are described in this section:

- Object Files
- Error Listing

### Object Files

After successful compilation, the Compiler generates an object file containing the target code as well as some debugging information. This file is written to the directory listed in the `OBJPATH`: Object File Path environment variable. If that variable contains more than one path, the object file is written in the first listed directory. If this variable is not set, the object file is written in the directory the source file was found. Object files always get the extension `*.o`.

### Error Listing

If the Compiler detects any errors, it does not create an object file. Rather, it creates an error listing file named `err.txt`. This file is generated in the directory where the source file was found (also see `ERRORFILE`: Error filename Specification environment variable).

If the Compiler's window is open, it displays the full path of all header files read. After successful compilation the number of code bytes generated and the number of global objects written to the object file are also displayed.

If the Compiler is started from an IDE (with `'%f'` given on the command line) or CodeWright (with `'%b%e'` given on the command line), this error file is not produced. Instead, it writes the error messages in a special format in a file called `EDOUT` using the Microsoft format by default. You may use the CodeWrights's *Find Next Error* command to display both the error positions and the error messages.

### Interactive Mode (Compiler window open)

If `ERRORFILE` is set, the Compiler creates a message file named as specified in this environment variable.

If `ERRORFILE` is not set, a default filename `err.txt` is generated in the current directory.

## Batch Mode (Compiler window not open)

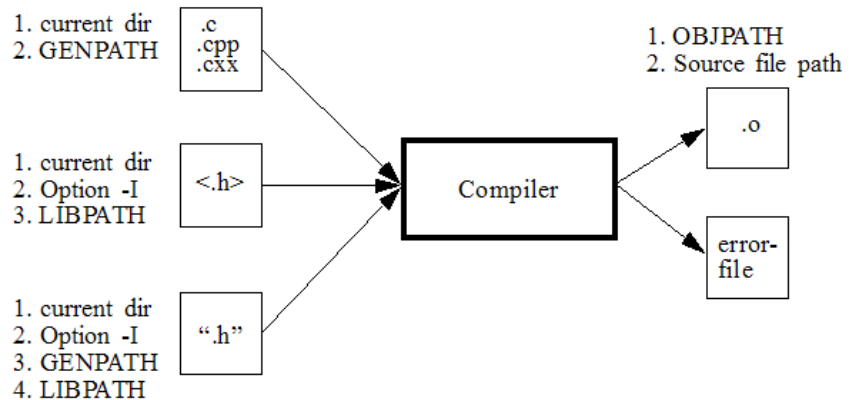
If `ERRORFILE` is set, the Compiler creates a message file named as specified in this environment variable.

If `ERRORFILE` is not set, a default file named `EDOUT` is generated in the current directory.

# File Processing

Figure 4.1 shows how file processing occurs with the Compiler:

**Figure 4.1 Files used with the Compiler**



## **Files**

*File Processing*

---

# Compiler Options

---

The major sections of this chapter are:

- Option Recommendation: Advice about the available compiler options
- Compiler Option Details: Description of the layout and format of the compiler command-line options that are covered in the remainder of the chapter.

The Compiler provides a number of Compiler options that control the Compiler's operation. Options consist of a minus sign or dash ( '-' ), followed by one or more letters or digits. Anything not starting with a dash or minus sign is the name of a source file to be compiled. You can specify Compiler options on the command line or in the `COMPOPTIONS` variable. Each Compiler option is specified only once per compilation.

Command line options are not case-sensitive, e.g., `-Li` is the same as `-li`.

---

**NOTE** It is not possible to coalesce options in different groups, e.g., `-Cc -Li` *cannot* be abbreviated by the terms `-Cci` or `-Ccli`!

---

Another way to set the Compiler options is to use the GUI (Figure 5.1).

---

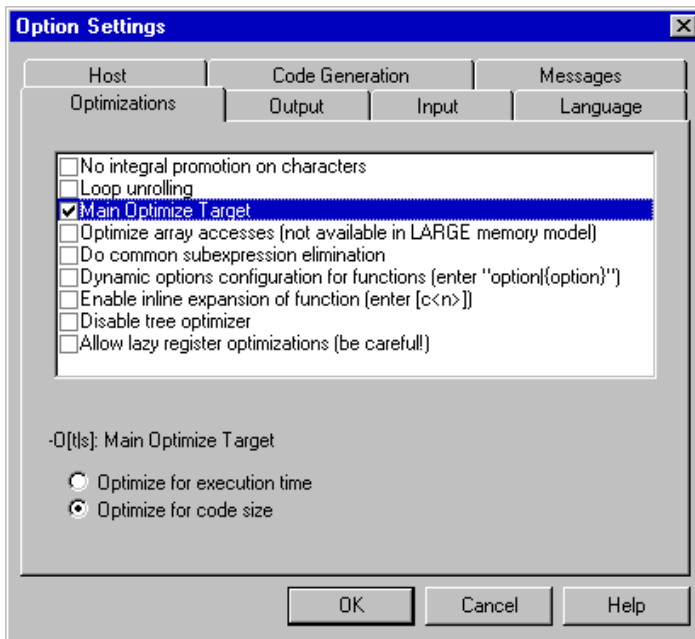
**NOTE** Do not use the `COMPOPTIONS` environment variable if the GUI is used. The Compiler stores the options in the `project.ini` file, not in the `default.env` file.

---

## Compiler Options

---

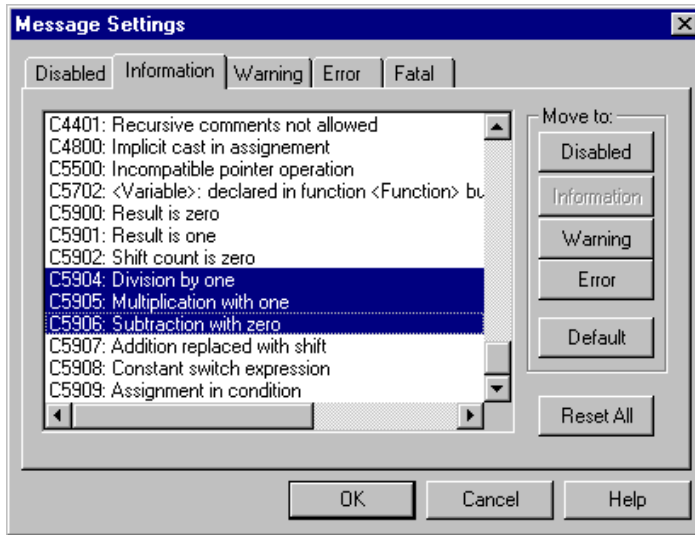
Figure 5.1 Option Settings dialog box



The Message Settings dialog box, shown in Figure 5.2, may also be used to move messages (-Wmsg options).



Figure 5.2 Message Settings Dialog



## Option Recommendation

Depending on the compiled sources, each Compiler optimization may have its advantages or disadvantages. The following are recommended:

- When using the HIWARE Object-file Format and the `-Cc: Allocate Constant Objects into ROM` compiler option, remember to specify `ROM_VAR` in the Linker parameter file.
- `-Wpd`: Error for Implicit Parameter Declaration
- `-Or`: Register Optimization whenever available or possible

The default configuration enables most optimizations in the Compiler. If they cause problems in your code (e.g., they make the code hard to debug), switch them off (these options usually have the `-On` prefix). Candidates for such optimizations are peephole optimizations.

Some optimizations may produce more code for some functions than for others (e.g., `-Oi`: Inlining or `-Cu`: Loop Unrolling). Try those options to get the best result for each.

To acquire the best results for each function, compile each module with the `-OdocF`: Dynamic Option Configuration for Functions option. An example for this option is `-OdocF=" -Or "`.

For compilers with the ICG optimization engine, the following option combination provides the best results:

-Ona -OdocF=" -Onca | -One | -Or "

## Compiler Option Details

### Option Groups

Compiler options are grouped by:

- HOST
- LANGUAGE
- OPTIMIZATIONS
- CODE GENERATION
- OUTPUT
- INPUT
- TARGET
- MESSAGES
- VARIOUS
- STARTUP

See Table 5.1.

A special group is the STARTUP group: The options in this group cannot be specified interactively; they can only be specified on the command line to start the tool.

**Table 5.1 Compiler option groups**

<b>Group</b>	<b>Description</b>
HOST	Lists options related to the host
LANGUAGE	Lists options related to the programming language (e.g., ANSI-C)
OPTIMIZATIONS	Lists optimization options
OUTPUT	Lists options related to the output files generation (which kind of file should be generated)
INPUT	Lists options related to the input file
CODE GENERATION	Lists options related to code generation (memory models, float format, ...)

**Table 5.1 Compiler option groups (continued)**

<b>Group</b>	<b>Description</b>
TARGET	Lists options related to the target processor
MESSAGES	Lists options controlling the generation of error messages
VARIOUS	Lists various options
STARTUP	Options which only are specified on tool startup

The group corresponds to the property sheets of the graphical option settings.

---

**NOTE** Not all command line options are accessible through the property sheets as they have a special graphical setting (e.g., the option to set the type sizes).

---

## Option Scopes

Each option has also a scope. See Table 5.2.

**Table 5.2 Option Scopes**

<b>Scope</b>	<b>Description</b>
Application	The option has to be set for all files (Compilation Units) of an application. A typical example is an option to set the memory model. Mixing object files will have unpredictable results.
Compilation Unit	This option is set for each compilation unit for an application differently. Mixing objects in an application is possible.
Function	The option may be set for each function differently. Such an option may be used with the option: <code>-OdocF= "&lt;option&gt;"</code> .
None	The option scope is not related to a specific code part. A typical example are the options for the message management.

The available options are arranged into different groups. A sheet is available for each of these groups. The content of the list box depends on the selected sheets.

## Option Detail Description

The remainder of this section describes each of the Compiler options available for the Compiler. The options are listed in alphabetical order. Each is divided into several sections listed in Table 5.3.

**Table 5.3 Compiler Option—Documentation Topics**

<b>Topic</b>	<b>Description</b>
Group	HOST, LANGUAGE, OPTIMIZATIONS, OUTPUT, INPUT, CODE GENERATION, MESSAGES or VARIOUS.
Scope	Application, Compilation Unit, Function or None
Syntax	Specifies the syntax of the option in an EBNF format
Arguments	Describes and lists optional and required arguments for the option
Default	Shows the default setting for the option
Defines	List of defines related to the compiler option
Pragma	List of pragmas related to the compiler option
Description	Provides a detailed description of the option and how to use it
Example	Gives an example of usage, and effects of the option where possible. compiler settings, source code and Linker PRM files are displayed where applicable. The example shows an entry in the <code>default.env</code> for a PC.
See also	Names related options

## Using Special Modifiers

With some options, it is possible to use special modifiers. However, some modifiers may not make sense for all options. This section describes those modifiers.

Table 5.4 lists the supported modifiers.

**Table 5.4 Compiler Option Modifiers**

<b>Modifier</b>	<b>Description</b>
%p	Path including file separator
%N	Filename in strict 8.3 format

**Table 5.4 Compiler Option Modifiers (continued)**

Modifier	Description
%n	Filename without extension
%E	Extension in strict 8.3 format
%e	Extension
%f	Path + filename without extension
%"	A double quote (") if the filename, the path or the extension contains a space
%'	A single quote (') if the filename, the path or the extension contains a space
%(ENV)	Replaces it with the contents of an environment variable
%%	Generates a single '%'

### Example

For the examples in Listing 5.1, the actual base filename for the modifiers is:  
 C:\Freescale\my\_demo\TheWholeThing.myExt.

**Listing 5.1 Examples of compiler option modifiers**

- (1) %p gives the path only with a file separator:  
 C:\Freescale\my\_demo\
- (2) %N results in the filename in 8.3 format (that is, the name with only 8 characters):  
 TheWhole
- (3) %n returns just the filename without extension:  
 TheWholeThing
- (4) %E gives the extension in 8.3 format (that is, the extension with only 3 characters)  
 myE
- (5) %e is used for the whole extension:  
 myExt
- (6) %f gives the path plus the filename:  
 C:\Freescale\my\_demo\TheWholeThing

## Compiler Options

### Compiler Option Details

---

(7) Because the path contains a space, using `% "` or `% '` is recommended: Thus, `%" %f%"` gives: (using double quotes)

```
"C:\Freescale\my demo\TheWholeThing"
```

(8) where `% '%f%'` gives: (using single quotes)

```
'C:\Freescale\my demo\TheWholeThing'
```

(9) `%(envVariable)` uses an environment variable. A file separator following after `%(envVariable)` is ignored if the environment variable is empty or does not exist. In other words, if `TEXTPATH` is set to: `TEXTPATH=C:\Freescale\txt`, `%(TEXTPATH)\myfile.txt` is replaced with:

```
C:\Freescale\txt\myfile.txt
```

(10) But if `TEXTPATH` does not exist or is empty, `%(TEXTPATH)\myfile.txt` is set to:  
`myfile.txt`

(11) A `%%` may be used to print a percent sign. Using `%e%%` gives:

```
myExt%
```

---

## **-!: filenames to DOS length**

### **Group**

INPUT

### **Scope**

Compilation Unit

### **Syntax**

- !

### **Arguments**

None

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

This option, called *cut*, is very useful when compiling files copied from an MS-DOS file system. filenames are clipped to DOS length (8 characters).

### **Listing 5.2 Example of the cut option, -!**

---

The cut option truncates the following include directive:

```
#include "mylongfilename.h"  
to:  
#include "mylongfi.h"
```

---

## **-AddIncl: Additional Include File**

### **Group**

INPUT

### **Scope**

Compilation Unit

### **Syntax**

```
-AddIncl"<fileName>"
```

### **Arguments**

<fileName>: name of file to be included

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

The specified file is included at the beginning of the compilation unit. It has the same effect as it would if written at the beginning of the compilation unit using double quotes (" . . "):

```
#include "my headerfile.h"
```

See Listing 5.3 for the `-AddIncl` compiler option to include the above header file.

### **Listing 5.3 Syntax example for including a header file.**

---

```
-AddIncl"my headerfile.h"
```

---



**See also**

-I: Include File Path compiler option

## **-Ansi: Strict ANSI**

### **Group**

LANGUAGE

### **Scope**

Function

### **Syntax**

`-Ansi`

### **Arguments**

None

### **Default**

None

### **Defines**

`__STDC__`

### **Pragmas**

None

### **Description**

The `-Ansi` option forces the Compiler to follow strict ANSI C language conversions. When `-Ansi` is specified, all non ANSI-compliant keywords (e.g., `__asm`, `__far` and `__near`) are not accepted by the Compiler, and the Compiler generates an error.

The ANSI-C compiler also does not allow C++ style comments (those started with `//`). To allow C++ comments, even with `-Ansi` set, the `-Cppc`: C++ Comments in ANSI-C compiler option must be set.

The `asm` keyword is also not allowed if `-Ansi` is set. To use inline assembly, even with `-Ansi` set use, `__asm` instead of `asm`.

The Compiler defines `__STDC__` as 1 if this option is set, or as 0 if this option is not set.

## **-Asr: It is assumed that HLI code saves written registers**

### **Group**

CODE GENERATION

### **Scope**

Function

### **Syntax**

-Asr

### **Arguments**

None

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

With this option set, the compiler assumes that registers touched in HLI are saved or restored in the HLI code as well. If this option is not set, the compiler will save or restore the H, X, and A registers.

### **Listing 5.4 Sample source code for the two following examples**

---

```
void bar(char);
char PORT;

void foo(void) {
    PORT = 4;
    asm {
        lda    #4
        sta    PORT
    }
}
```

## Compiler Options

### Compiler Option Details

---

```
    }  
    bar(4);  
}
```

---

#### Listing 5.5 Without the -Asr option set, we get:

---

```
4:      PORT = 4;  
0000 a604          LDA    #4  
0002 c70000       STA    PORT  
5:      __asm {  
6:      lda    #4  
0005 a604          LDA    #4  
7:      sta    PORT  
0007 c70000       STA    PORT  
8:      }  
9:      bar(4);  
000a a604          LDA    #4  
000c cc0000       JMP    bar
```

---

With the `-Asr` option set (Listing 5.6), the compiler can assume that the A register is still the same as before the `__asm` block. However, in our example we do NOT save or restore the A register, so the code will be incorrect.

#### Listing 5.6 Without the -Asr option set, we get:

---

```
4:      PORT = 4;  
0000 a604          LDA    #4  
0002 c70000       STA    PORT  
5:      __asm {  
6:      lda    #4  
0005 a604          LDA    #4  
7:      sta    PORT  
0007 c70000       STA    PORT  
8:      }  
9:      bar(4);  
000a cc0000       JMP    bar
```

---

## -BfaB: Bitfield Byte Allocation

### Group

CODE GENERATION

### Scope

Function

### Syntax

-BfaB (MS | LS)

### Arguments

MS: Most significant bit in byte first (left to right)

LS: Least significant bit in byte first (right to left)

### Default

HC08: -BfaBLS

### Defines

```
__BITFIELD_MSWORD_FIRST__  
__BITFIELD_LSWORD_FIRST__  
__BITFIELD_MSBYTE_FIRST__  
__BITFIELD_LSBYTE_FIRST__  
__BITFIELD_MSBIT_FIRST__  
__BITFIELD_LSBIT_FIRST__
```

### Pragmas

None

### Description

Normally, bits in byte bitfields are allocated from the least significant bit to the most significant bit. This produces less code overhead if a byte bitfield is allocated only partially.

### Example

Listing 5.7 uses the default condition and uses the three least significant bits.

## Compiler Options

### Compiler Option Details

---

#### Listing 5.7 Example struct used in Listing 5.8

---

```
struct {unsigned char b: 3; } B;
// the default is using the 3 least significant bits
```

---

This allows just a mask operation without any shift to access the bitfield.

To change this allocation order, you can use the `-BfaBMS` or `-BfaBLS` options, shown in the Listing 5.8.

#### Listing 5.8 Examples of changing the bitfield allocation order

---

```
struct {
    char b1:1;
    char b2:1;
    char b3:1;
    char b4:1;
    char b5:1;
} myBitfield;
```

```
7                                     0
-----
|b1|b2|b3|b4|b5|####| (-BfaBMS)
```

```
7                                     0
-----
|####|b5|b4|b3|b2|b1| (-BfaBLS)
```

---

### See also

Bitfield Allocation

## **-BfaGapLimitBits: Bitfield Gap Limit**

### **Group**

CODE GENERATION

### **Scope**

Function

### **Syntax**

`-BfaGapLimitBits<number>`

### **Arguments**

`<number>`: positive number specifying the maximum number of bits for a gap

### **Default**

HC08: 0

### **Defines**

None

### **Pragmas**

None

### **Description**

The bitfield allocation tries to avoid crossing a byte boundary whenever possible. To achieve optimized accesses, the compiler may insert some padding or gap bits. This option enables you to affect the maximum number of gap bits allowed.

### **Example**

In the example in Listing 5.9, it is assumed that you have specified a 3-bit gap.

#### **Listing 5.9 Bitfield allocation**

---

```
struct {  
    unsigned char a: 7;  
    unsigned char b: 5;  
    unsigned char c: 4;  
} B;
```

---

## Compiler Options

### Compiler Option Details

---

The compiler allocates `struct B` with 3 bytes. First the compiler allocates the 7 bits of `a`. Then the compiler tries to allocate the 5 bits of `b`, but this would cross a byte boundary. Because the gap of 1 bit is smaller than the specified gap of 3 bits, `b` is allocated in the next byte. Then the allocation starts for `c`. After the allocation of `b`, there are 3 bits left. Because the gap is 3 bits, `c` is allocated in the next byte. If the maximum gap size were specified to 0, all bits would be allocated in two bytes.

Listing 5.10 specifies a maximum size of three bits for a gap.

#### Listing 5.10 Example where the maximum number of gap bits is 3

---

```
-BfaGapLimitBits3
```

---

#### See also

Bitfield Allocation



## **-BfaTSR: Bitfield Type Size Reduction**

### **Group**

CODE GENERATION

### **Scope**

Function

### **Syntax**

`-BfaTSR(ON|OFF)`

### **Arguments**

ON: Enable Type Size Reduction

OFF: Disable Type Size Reduction

### **Default**

HC08: `-BfaTSRon`

### **Defines**

`__BITFIELD_TYPE_SIZE_REDUCTION__`  
`__BITFIELD_NO_TYPE_SIZE_REDUCTION__`

### **Pragmas**

None

### **Description**

This option is possible whether or not the compiler uses type size-reduction for bitfields. Type-size reduction means that the compiler can reduce the type of an `int` bitfield to a `char` bitfield if it fits into a character. This allows the compiler to allocate memory only for one byte instead of for an integer.

### **Examples**

Listing 5.11 and Listing 5.12 demonstrate the effects of `-BfaTSRoff` and `-BfaTSRon`, respectively.

## Compiler Options

### Compiler Option Details

---

#### Listing 5.11 -BfaTSRoff.

---

```
struct{
    long b1:4;
    long b2:4;
} myBitfield;

31                                     7 3 0
-----
|#####|b2|b1| -BfaTSRoff
-----
```

---

#### Listing 5.12 -BfaTSRon

---

```
7 3 0
-----
|b2 | b1 | -BfaTSRon
-----
```

---

### Example

-BfaTSRon

### See also

Bitfield Type Reduction

## -Cc: Allocate Constant Objects into ROM

### Group

OUTPUT

### Scope

Compilation Unit

### Syntax

-Cc

### Arguments

None

### Default

None

### Defines

None

### Pragmas

#pragma INTO\_ROM: Put Next Variable Definition into ROM

### Description

In the HIWARE Object-file Format, variables declared as `const` are treated just like any other variable, unless the `-Cc` command line option was used. In that circumstance, the `const` objects are put into the `ROM_VAR` segment which is then assigned to a ROM section in the Linker parameter file (please see the *Linker* section in the *Build Tool Utilities* manual).

The Linker prepares no initialization for objects allocated into a read-only section. The startup code does not have to copy the constant data.

You may also put variables into the `ROM_VAR` segment by using the segment pragma (please see the *Linker* section in the *Build Tool Utilities* manual).

With the `#pragma CONST_SECTION` for constant-segment allocation, variables declared as `const` are allocated in this segment.

If the current data segment is not the default segment, `const` objects in that user-defined segment are not allocated in the `ROM_VAR` segment but remain in the

## Compiler Options

### Compiler Option Details

---

segment defined by the user. If that data segment happens to contain *only* `const` objects, it may be allocated in a ROM memory section (refer to the *Linker* section in the *Build Tool Utilities* manual for more information).

---

**NOTE** This option is useful only for HIWARE object-file formats. In the ELF/DWARF object-file format, constants are allocated into the `".rodata"` section.

---

---

**NOTE** The Compiler uses the default addressing mode for the constants specified by the memory model.

---

### Example

Listing 5.13 shows how the `-Cc` compiler option affects the `SECTIONS` segment of a PRM file (HIWARE object-file format only).

#### Listing 5.13 Constant objects into ROM

---

```
SECTIONS
  MY_ROM READ_ONLY      0x1000 TO 0x2000
PLACEMENT
  DEFAULT_ROM, ROM_VAR INTO MY_ROM
```

---

### See also

- Segmentation
- Linker Manual
- F (-Fh, -F1, -F1o, -F2, -F2o, -F6, or -F7): Object-File Format option
- `#pragma INTO_ROM`: Put Next Variable Definition into ROM

## **-Ccx: Cosmic Compatibility Mode for Space Modifiers and Interrupt Handlers**

### **Group**

LANGUAGE

### **Scope**

Compilation Unit

### **Syntax**

-Ccx

### **Arguments**

None

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

This option allows Cosmic style @near, @far and @tiny space modifiers as well as @interrupt in your C code. The -ANSI option must be switched off. It is not necessary to remove the Cosmic space modifiers from your application code. There is no need to place the objects to sections addressable by the Cosmic space modifiers.

The following is done when a Cosmic modifier is parsed:

- The objects declared with the space modifier are always allocated in a special Cosmic compatibility (\_CX...) section (regardless which section pragma is set) depending on the space modifier, on the const qualifier or if it is a function or a variable:

## Compiler Options

### Compiler Option Details

---

- Space modifiers on the left hand side of a pointer declaration specify the pointer type and `pointer` size, depending on the target.

See the example in Listing 5.14 for a PRM file about how to place the sections mentioned in the Table 5.5.

**Table 5.5 Cosmic Modifier Handling**

Definition	Placement to <code>_CX</code> section
<code>@tiny int my_var</code>	<code>_CX_DATA_TINY</code>
<code>@near int my_var</code>	<code>_CX_DATA_NEAR</code>
<code>@far int my_var</code>	<code>_CX_DATA_FAR</code>
<code>const @tiny int my_cvar</code>	<code>_CX_CONST_TINY</code>
<code>const @near int my_cvar</code>	<code>_CX_CONST_NEAR</code>
<code>const @far int my_cvar</code>	<code>_CX_CONST_FAR</code>
<code>@tiny void my_fun(void)</code>	<code>_CX_CODE_TINY</code>
<code>@near void my_fun(void)</code>	<code>_CX_CODE_NEAR</code>
<code>@far void my_fun(void)</code>	<code>_CX_CODE_FAR</code>
<code>@interrupt void my_fun(void)</code>	<code>_CX_CODE_INTERRUPT</code>

For further information about porting applications from Cosmic to CodeWarrior please refer to the technical note TN 234. Table 5.6 gives an overview how space modifiers are mapped for the HC08:

**Table 5.6 Cosmic Space Modifier Mapping for HC08**

Definition	Keyword Mapping
<code>@tiny</code>	<code>__near</code>
<code>@near</code>	<code>__far</code>
<code>@far</code>	<code>__far</code>

See Listing 5.14 for an example of the `-Ccx` compiler option.

**Listing 5.14 Cosmic Space Modifiers**

---

```
volatile @tiny char tiny_ch;
extern @far const int table[100];
static @tiny char * @near ptr_tab[10];
typedef @far int (*@far funptr)(void);
funptr my_fun; /* banked and __far calling conv. */

char @tiny *tptr = &tiny_ch;
char @far *fptr = (char @far *)&tiny_ch;
```

Example for a prm file:  
(16- and 24-bit addressable ROM;  
8-, 16- and 24-bit addressable RAM)

```
SEGMENTS
MY_ROM    READ_ONLY    0x2000    TO 0x7FFF;
MY_BANK   READ_ONLY    0x508000 TO 0x50BFFF;
MY_ZP     READ_WRITE   0xC0     TO 0xFF;
MY_RAM    READ_WRITE   0xC000    TO 0xCFFF;
MY_DBANK  READ_WRITE   0x108000 TO 0x10BFFF;
END
PLACEMENT
DEFAULT_ROM, ROM_VAR,
_CX_CODE_NEAR, _CX_CODE_TINY, _CX_CONST_TINY,
_CX_CONST_NEAR          INTO MY_ROM;
_CX_CODE_FAR, _CX_CONST_FAR INTO MY_BANK;
DEFAULT_RAM, _CX_DATA_NEAR INTO MY_RAM;
_CX_DATA_FAR          INTO MY_DBANK;
_ZEROPAGE, _CX_DATA_TINY  INTO MY_ZP;
END
```

---

**See also**

Cosmic Manuals, Linker Manual, TN 234

## **-Ci: Tri- and Bigraph Support**

### **Group**

LANGUAGE

### **Scope**

Function

### **Syntax**

-Ci

### **Arguments**

None

### **Default**

None

### **Defines**

\_\_\_TRIGRAPHS\_\_\_

### **Pragmas**

None

### **Description**

If certain tokens are not available on your keyboard, they are replaced with keywords as shown in Table 5.7.

**Table 5.7 Keyword Alternatives for Unavailable Tokens**

<b>Bigraph</b>		<b>Trigraph</b>		<b>Additional Keyword</b>	
<%	}	??=	#	and	&&
%>	}	??/	\	and_eq	&=
<:	[	??'	^	bitand	&
:>	]	??(	[	bitor	
%.:	#	??)	]	compl	~



**Table 5.7 Keyword Alternatives for Unavailable Tokens (*continued*)**

Bigraph		Trigraph		Additional Keyword	
%.%:	##	??!		not	!
		??<	{	or	
		??>	}	or_eq	=
		??~	~	xor	^
				xor_eq	^=
				not_eq	!=

---

**NOTE** Additional keywords are not allowed as identifiers if this option is enabled.

---

### Example

-Ci

The example in Listing 5.15 shows the use of trigraphs, bigraphs and the additional keywords with the corresponding ‘normal’ C-source.

**Listing 5.15 Trigraphs, Bigraphs, and Additional Keywords**

```
int Trigraphs(int argc, char * argv??(??)) ??<
    if (argc<1 ??!??! *argv??(1??)=='??/0') return 0;
    printf("Hello, %s??/n", argv??(1??));
??>
%:define TEST_NEW_THIS 5
%:define cat(a,b) a%:%:b
??=define arraycheck(a,b,c) a??(i??) ??!??! b??(i??)
int i;
int cat(a,b);
char a<:10:>;
char b<:10:>;

void Trigraph2(void) <%
    if (i and ab) <%
        i and_eq TEST_NEW_THIS;
        i = i bitand 0x03;
        i = i bitor 0x8;
        i = compl i;
        i = not i;
    %> else if (ab or i) <%
```

## Compiler Options

### Compiler Option Details

---

```
    i or_eq 0x5;
    i = i xor 0x12;
    i xor_eq 99;
%> else if (i not_eq 5) <%
    cat(a,b) = 5;
    if (a??(i??) || b[i])<%%>
    if (arraycheck(a,b,i)) <%
        i = 0;
    %>
%>
%>
/* is the same as ... */
int Trigraphs(int argc, char * argv[]) {
    if (argc<1 || *argv[1]!='\0') return 0;
    printf("Hello, %s\n", argv[1]);
}

#define TEST_NEW_THIS 5
#define cat(a,b) a##b
#define arraycheck(a,b,c) a[i] || b[i]
int i;
int cat(a,b);
char a[10];
char b[10];

void Trigraph2(void){
    if (i && ab) {
        i &= TEST_NEW_THIS;
        i = i & 0x03;
        i = i | 0x8;
        i = ~i;
        i = !i;
    } else if (ab || i) {
        i |= 0x5;
        i = i ^ 0x12;
        i ^= 99;
    } else if (i != 5) {
        cat(a,b) = 5;
        if (a[i] || b[i]){}
        if (arraycheck(a,b,i)) {
            i = 0;
        }
    }
}
}
```

---

## **-Cni: No Integral Promotion**

### **Group**

OPTIMIZATIONS

### **Scope**

Function

### **Syntax**

-Cni

### **Arguments**

None

### **Default**

None

### **Defines**

\_\_\_CNI\_\_\_

### **Pragmas**

None

### **Description**

Enhances code density of character operations by omitting integral promotion. This option enables a non ANSI-C compliant behavior.

In ANSI-C operations with data types, anything smaller than int must be promoted to int (integral promotion). With this rule, adding two unsigned character variables results in a zero-extension of each character operand, and then adding them back in as int operands. If the result must be stored back into a character, this integral promotion is not necessary. When this option is set, promotion is avoided where possible.

The code size may be decreased if this option is set because operations may be performed on a character base instead of an integer base.

The -Cni option enhances character operation code density by omitting integral promotion.

Consider the following:

## Compiler Options

### Compiler Option Details

---

In most expressions, ANSI-C requires char type variables to be extended to the next larger type int, which is required to be at least 16-bit in size by the ANSI standard.

The `-Cni` option suppresses this ANSI-C behavior and thus allows 'characters' and 'character-sized constants' to be used in expressions. This option does not conform to ANSI standards. Code compiled with this option is not portable.

The ANSI standard requires that 'old style declarations' of functions using the `char` parameter (Listing 5.16) be extended to `int`. The `-Cni` option disables this extension and saves additional RAM.

### Example

See Listing 5.16.

#### Listing 5.16 Definition of an 'old style function using a char parameter.

---

```
old_style_func (a, b, c)
    char a, b, c;
{
    ...
}
```

---

The space reserved for a, b, and c is just one byte each, instead of two.

For expressions containing different types of variables, the following conversion rules apply:

If both variables are of type `signed char`, the expression is evaluated `signed`.

If one of two variables is of type `unsigned char`, the expression is evaluated `unsigned`, regardless of whether the other variable is of type `signed` or `unsigned char`.

If one operand is of another type than `signed` or `unsigned char`, the usual ANSI-C arithmetic conversions are applied.

If constants are in the character range, they are treated as characters. Remember that the `char` type is `signed` and applies to the constants `-128` to `127`. All constants greater than `127`, i.e., `128`, `129` ... are treated as integer. If you want them treated as characters, they must be casted (Listing 5.17).

#### Listing 5.17 Casting integers to signed char

---

```
signed char a, b;
if (a > b * (signed char)129)
```

---

---

**NOTE** This option is ignored with the `-Ansi` Compiler switch active.

---

---

**NOTE** With this option set, the code that is generated does not conform to the ANSI standard. In other words: the code generated is wrong if you apply the ANSI standard as reference. Using this option is not recommended in most cases.

---

## **-Cppc: C++ Comments in ANSI-C**

### **Group**

LANGUAGE

### **Scope**

Function

### **Syntax**

-Cppc

### **Arguments**

None

### **Default**

By default, the Compiler does not allow C++ comments if the -Ansi: Strict ANSI compiler option is set.

### **Defines**

None

### **Pragmas**

None

### **Description**

The -Ansi option forces the compiler to conform to the ANSI-C standard. Because a strict ANSI-C compiler rejects any C++ comments (started with //), this option may be used to allow C++ comments (Listing 5.18).

### **Listing 5.18 Using -Cppc to allow C++ comments**

---

```
-Cppc
/* This allows the code containing C++ comments to be compiled with
   the -Ansi option set */
void foo(void) // this is a C++ comment
```

---

**See also**

-Ansi: Strict ANSI compiler option

## **-Cq: Propagate const and volatile qualifiers for structs**

### **Group**

LANGUAGE

### **Scope**

Application

### **Syntax**

-Cq

### **Arguments**

None

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

This option propagates `const` and `volatile` qualifiers for structures. That means, if all members of a structure are constant, the structure itself is constant as well. The same happens with the `volatile` qualifier. If the structure is declared as constant or volatile, all members are constant or volatile. Consider the following example.

### **Example**

The source code in Listing 5.19 declares two structs, each of which has a `const` member.

#### **Listing 5.19**

---

```
struct {  
    const field;
```



```
} s1, s2;

void foo(void) {
    s1 = s2; // struct copy
    s1.field = 3; // error: modifiable lvalue expected
}
```

---

In the above example, the field in the struct is constant, but not the struct itself. Thus the struct copy 's1 = s2' is legal, even if the field of the struct is constant. But, a write access to the struct field causes an error message. Using the `-Cq` option propagates the qualification (`const`) of the fields to the whole struct or array. In the above example, the struct copy would cause an error message.

## **-Cs08: Generate Code for HCS08**

### **Group**

CODE GENERATION

### **Scope**

Compilation Unit

### **Syntax**

-Cs08

### **Arguments**

None

### **Default**

None

### **Defines**

\_\_HCS08\_\_

### **Pragmas**

None

### **Description**

Generates code for the HCS08 family. In addition, allows all new HCS08 operation-operand combinations in the HLI.

### **Example**

\_\_asm LDHX 2,X;

## **-CswMaxLF: Maximum Load Factor for Switch Tables**

### **Group**

CODE GENERATION

### **Scope**

Function

### **Syntax**

`-CswMaxLF<number>`

### **Arguments**

`<number>`: a number in the range of 0 to 100 denoting the maximum load factor.

### **Default**

Backend-dependent.

### **Defines**

None

### **Pragmas**

None

### **Description**

Allows changing the default strategy of the Compiler to use tables for switch statements.

---

**NOTE** This option is only available if the compiler supports switch tables.

---

Normally the Compiler uses a table for switches with more than about 8 labels if the table is filled between 80% (minimum load factor of 80) and 100% (maximum load factor of 100). If there are not enough labels for a table or the table is not filled, a branch tree is generated (tree of if-else-if-else). This branch tree is like an ‘unrolled’ binary search in a table which quickly evaluates the associated label for a switch expression.

Using a branch tree instead of a table improves code execution speed, but may increase code size. In addition, because the branch tree itself uses no special

## Compiler Options

### Compiler Option Details

---

runtime routine for switch expression evaluation, debugging may be more seamless.

Specifying a load factor means that tables are generated in specific ‘fuel’ status:

#### Listing 5.20

---

```
switch(i) {
  case 0: ...
  case 1: ...
  case 2: ...
  case 3: ...
  case 4: ...
  // case 5: ...
  case 6: ...
  case 7: ...
  case 8: ...
  case 9: ...
  default
}
```

---

The above table is filled to 90% (labels for ‘0’ to ‘9’, except for ‘5’). Assumed that the minimum load factor is set to 50% and setting the maximum load factor for the above case to 80%, a branch tree is generated instead a table. But setting the maximum load factor to 95% will produce a table.

To guarantee that tables are generated for switches with full tables only, set the table minimum and maximum load factors to 100:

```
-CswMinLF100 -CswMaxLF100 .
```

#### See also

##### Compiler options:

- -CswMinLB: Minimum Number of Labels for Switch Tables
- Option -CswMinSLB: Minimum Number of Labels for Search Switch Tables
- -CswMinLF: Minimum Load Factor for Switch Tables

## **-CswMinLB: Minimum Number of Labels for Switch Tables**

### **Group**

CODE GENERATION

### **Scope**

Function

### **Syntax**

`-CswMinLB<number>`

### **Arguments**

`<number>`: a positive number denoting the number of labels.

### **Default**

Backend-dependent

### **Defines**

None

### **Pragmas**

None

### **Description**

This option allows changing the default strategy of the Compiler using tables for switch statements.

---

**NOTE** This option is only available if the compiler supports switch tables.

---

Normally the Compiler uses a table for switches with more than about 8 labels (case entries) (actually this number is highly backend-dependent). If there are not enough labels for a table, a branch tree is generated (tree of if-else-if-else). This branch tree is like an ‘unrolled’ binary search in a table which evaluates very fast the associated label for a switch expression.

Using a branch tree instead of a table may increase the code execution speed, but it probably increases the code size. In addition, because the branch tree itself uses no special runtime routine for switch expression evaluation, debugging may be much easier.

## Compiler Options

### *Compiler Option Details*

---

To disable any tables for switch statements, just set the minimum number of labels needed for a table to a high value (e.g., 9999):

```
-CswMinLB9999 -CswMinSLB9999.
```

When disabling simple tables it usually makes sense also to disable search tables with the `-CswMinSLB` option.

### **See also**

#### **Compiler options:**

- `-CswMinLF`: Minimum Load Factor for Switch Tables
- `-CswMinSLB`: Minimum Number of Labels for Search Switch Tables
- `-CswMaxLF`: Maximum Load Factor for Switch Tables

## **-CswMinLF: Minimum Load Factor for Switch Tables**

### **Group**

CODE GENERATION

### **Scope**

Function

### **Syntax**

`-CswMinLF<number>`

### **Arguments**

`<number>`: a number in the range of 0 – 100 denoting the minimum load factor

### **Default**

Backend-dependent

### **Defines**

None

### **Pragmas**

None

### **Description**

Allows the Compiler to use tables for switch statements.

---

**NOTE** This option is only available if the compiler supports switch tables.

---

Normally the Compiler uses a table for switches with more than about 8 labels and if the table is filled between 80% (minimum load factor of 80) and 100% (maximum load factor of 100). If there are not enough labels for a table or the table is not filled, a branch tree is generated (tree of if-else-if-else). This branch tree is like an ‘unrolled’ binary search in a table which quickly evaluates the associated label for a switch expression.

Using a branch tree instead of a table improves code execution speed, but may increase code size. In addition, because the branch tree itself uses no special runtime routine for switch expression evaluation, debugging is more seamless.

Specifying a load factor means that tables are generated in specific ‘fuel’ status:

### Listing 5.21

---

```
switch(i) {
  case 0: ...
  case 1: ...
  case 2: ...
  case 3: ...
  case 4: ...
  // case 5: ...
  case 6: ...
  case 7: ...
  case 8: ...
  case 9: ...
  default
}
```

---

The above table is filled to 90% (labels for '0' to '9', except for '5'). Assuming that the maximum load factor is set to 100% and the minimum load factor for the above case is set to 90%, this still generates a table. But setting the minimum load factor to 95% produces a branch tree.

To guarantee that tables are generated for switches with full tables only, set the minimum and maximum table load factors to 100: `-CswMinLF100`  
`-CswMaxLF100`.

### See also

#### Compiler options:

- `-CswMinLB`: Minimum Number of Labels for Switch Tables
- `-CswMinSLB`: Minimum Number of Labels for Search Switch Tables
- `-CswMaxLF`: Maximum Load Factor for Switch Tables



## **-CswMinSLB: Minimum Number of Labels for Search Switch Tables**

### **Group**

CODE GENERATION

### **Scope**

Function

### **Syntax**

`-CswMinSLB<number>`

### **Arguments**

`<number>`: a positive number denoting the number of labels

### **Default**

Backend-dependent

### **Defines**

None

### **Pragmas**

None

### **Description**

Allows the Compiler to use tables for switch statements.

---

**NOTE** This option is only available if the compiler supports search tables.

---

Switch tables are implemented in different ways. When almost all case entries in some range are given, a table containing only branch targets is used. Using such a dense table is efficient because only the correct entry is accessed. When large holes exist in some areas, a table form can still be used.

But now the case entry and its corresponding branch target are encoded in the table. This is called a search table. A complex runtime routine must be used to access a search table. This routine checks all entries until it finds the matching one. Search tables execute slowly.

Using a search table improves code density, but the execution time increases. Every time an entry in a search table must be found, all previous entries must be

## Compiler Options

### *Compiler Option Details*

---

checked first. For a dense table, the right offset is computed and accessed. In addition, note that all backends implement search tables (if at all) by using a complex runtime routine. This may make debugging more complex.

To disable search tables for switch statements, set the minimum number of labels needed for a table to a high value (e.g., 9999): `-CswMinSLB9999`.

### **See also**

#### **Compiler options:**

- `-CswMinLB`: Minimum Number of Labels for Switch Tables
- `-CswMinLF`: Minimum Load Factor for Switch Tables
- `-CswMaxLF`: Maximum Load Factor for Switch Tables

## **-Cu: Loop Unrolling**

### **Group**

OPTIMIZATIONS

### **Scope**

Function

### **Syntax**

`-Cu [=i<number>]`

### **Arguments**

`<number>`: number of iterations for unrolling, between 0 and 1024

### **Default**

None

### **Defines**

None

### **Pragmas**

`#pragma LOOP_UNROLL`: Force Loop Unrolling

`#pragma NO_LOOP_UNROLL`: Disable Loop Unrolling

### **Description**

Enables loop unrolling with the following restrictions:

- Only simple `for` statements are unrolled, e.g.,  
`for (i=0; i<10; i++)`
- Initialization and test of the loop counter must be done with a constant.
- Only `<`, `>`, `<=`, `>=` are permitted in a condition.
- Only `++` or `--` are allowed for the loop variable increment or decrement.
- The loop counter must be integral.
- No change of the loop counter is allowed within the loop.
- The loop counter must not be used on the left side of an assignment.
- No address operator (`&`) is allowed on the loop counter within the loop.

## Compiler Options

### Compiler Option Details

---

- Only small loops are unrolled:
- Loops with few statements within the loop.
- Loops with fewer than 16 increments or decrements of the loop counter.  
The bound may be changed with the optional argument = `i<number>`.  
The `-Cu=i20` option unrolls loops with a maximum of 20 iterations.

### Examples

#### Listing 5.22 for Loop

---

```
-Cu
int i, j;
j = 0;
for (i=0; i<3; i++) {
    j += i;
}
```

---

With the `-Cu` compiler option given, the Compiler issues an information message '*Unrolling loop*' and transforms this loop as shown in Listing 5.23.:

#### Listing 5.23 Transformation of the Loop in Listing 5.22

---

```
j += 1;
j += 2;
i = 3;
```

---

The Compiler also transforms some special loops, i.e., loops with a constant condition or loops with only one pass:

#### Listing 5.24 Example for a loop with a constant condition

---

```
for (i=1; i>3; i++) {
    j += i;
}
```

---

The Compiler issues an information message '*Constant condition found, removing loop*' and transforms the loop into a simple assignment

```
i=1;
```

because the loop body is never executed.

**Listing 5.25 Example for a loop with only one pass**

---

```
for (i=1; i<2; i++) {  
    j += i;  
}
```

---

The Compiler issues a warning '*Unrolling loop*' and transforms the `for` loop into

```
j += 1;  
i = 2;
```

because the loop body is executed only once.

## **-Cx: No Code Generation**

### **Group**

CODE GENERATION

### **Scope**

Compilation Unit

### **Syntax**

-Cx

### **Arguments**

None

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

The -Cx Compiler option disables the code generation process of the Compiler. No object code is generated, though the Compiler does perform a syntactical check of the source code. This allows a quick test if the Compiler accepts the source without errors.

## **-D: Macro Definition**

### **Group**

LANGUAGE

### **Scope**

Compilation Unit

### **Syntax**

```
-D<identifier>[=<value>]
```

### **Arguments**

<identifier>: identifier to be defined

<value>: value for <identifier>, anything except "-" and blank

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

The Compiler allows the definition of a macro on the command line. The effect is the same as having a `#define` directive at the very beginning of the source file.

### **Listing 5.26 DEBUG macro definition.**

---

```
-DDEBUG=0
```

This is the same as writing:

```
#define DEBUG 0
```

in the source file.

---

## Compiler Options

### Compiler Option Details

---

If you need strings with blanks in your macro definition, there are two ways:  
escape sequence or double quotes:

```
-dPath="Path\40with\40spaces"
```

```
-d"Path=" "Path with spaces" "
```

---

**NOTE** Blanks are *not* allowed after the `-D` option – the first blank terminates this option. Also, macro parameters are not supported.

---



## **-Ec: Conversion from 'const T\*' to 'T\*'**

### **Group**

LANGUAGE

### **Scope**

Function

### **Syntax**

-Ec

### **Arguments**

None

### **Default**

None

### **Description**

If this non-ANSI compliant extension is enabled, a pointer to a constant type is treated like a pointer to the non-constant equivalent of the type. Earlier Compilers did not check a store to a constant object through a pointer. This option is useful if some older source has to be compiled.

### **Listing 5.27 Examples**

---

```
void f() {
    int *i;
    const int *j;
    i=j; /* C++ illegal, but with -Ec ok! */
}

struct A {
    int i;
};

void g() {
    const struct A *a;
    a->i=3; /* ANSI C/C++ illegal, but with -Ec ok! */
}

void h() {
```

## Compiler Options

### Compiler Option Details

---

```
const int *i;
*i=23; /* ANSI-C/C++ illegal, but with -Ec ok! */
}
```

---

#### Defines

None

#### Pragmas

None

#### Listing 5.28 Example

---

```
-Ec
void foo(const int *p){
    *p = 0; // some Compilers do not issue an error
}
```

---

## -Eencrypt: Encrypt Files

### Group

OUTPUT

### Scope

Compilation Unit

### Syntax

```
-Eencrypt [= <filename> ]
```

### Arguments

<filename>: The name of the file to be generated

It may contain special modifiers (see Using Special Modifiers).

### Default

The default filename is %f.e%. A file named 'foo.c' creates an encrypted file named 'foo.ec'.

### Description

All files passed together with this option are encrypted using the given key with the -Ekey: Encryption Key option.

---

**NOTE** This option is only available or operative with a license for the following feature: HIxxxx30 where xxxx is the feature number of the compiler for a specific target.

---

### Defines

None

### Pragmas

None

### Example

```
foo.c foo.h -Ekey1234567 -Eencrypt=%n.e%
```

encrypts the 'foo.c' file using the 1234567 key to the 'foo.ec' file and the 'foo.h' file to the 'foo.eh' file.

## Compiler Options

### *Compiler Option Details*

---

The encrypted `foo.ec` and `foo.eh` files may be passed to a client. The client is able to compile the encrypted files without the key compiling the following file:

`foo.ec`

### **See also**

-Ekey: Encryption Key

## **-Ekey: Encryption Key**

### **Group**

OUTPUT

### **Scope**

Compilation Unit

### **Syntax**

`-Ekey<keyNumber>`

### **Arguments**

`<keyNumber>`

### **Default**

The default encryption key is '0'. Using this default is not recommended.

### **Description**

This option is used to encrypt files with the given key number (`-Eencrypt` option).

---

**NOTE** This option is only available or operative with a license for the following feature: HIxxxx30 where xxxx is the feature number of the compiler for a specific target.

---

### **Defines**

None

### **Pragmas**

None

### **Example**

```
foo.c -Ekey1234567 -Eencrypt=%n.e%e  
encrypts the 'foo.c' file using the 1234567 key.
```

### **See also**

`-Eencrypt`: Encrypt Files

## **-Env: Set Environment Variable**

### **Group**

HOST

### **Scope**

Compilation Unit

### **Syntax**

`-Env<Environment Variable>=<Variable Setting>`

### **Arguments**

`<Environment Variable>`: Environment variable to be set

`<Variable Setting>`: Setting of the environment variable

### **Default**

None

### **Description**

This option sets an environment variable. This environment variable may be used in the maker, or used to overwrite system environment variables.

### **Defines**

None

### **Pragmas**

None

### **Example**

```
-EnvOBJPATH=\sources\obj
```

This is the same as:

```
OBJPATH=\sources\obj
```

in the `default.env` file.

Use the following syntax to use an environment variable that uses filenames with spaces:

```
-Env "OBJPATH=\program files"
```

**See also**

Environment

---

## **-F (-Fh, -F1, -F1o, -F2, -F2o, -F6, or -F7): Object-File Format**

### **Group**

OUTPUT

### **Scope**

Application

### **Syntax**

`-F (h|1|1o|2|2o|6|7)`

### **Arguments**

h: HIWARE object-file format  
1: ELF/DWARF 1.1 object-file format  
1o: compatible ELF/DWARF 1.1 object-file format  
2: ELF/DWARF 2.0 object-file format  
2o: compatible ELF/DWARF 2.0 object-file format  
6: strict HIWARE V2.6 object-file format  
7: strict HIWARE V2.7 object-file format

---

**NOTE** Not all object-file formats may be available for a target.

---

### **Default**

`-F2`

### **Defines**

`__HIWARE_OBJECT_FILE_FORMAT__`  
`__ELF_OBJECT_FILE_FORMAT__`

### **Pragmas**

None

### **Description**

The Compiler writes the code and debugging info after compilation into an object file.



The Compiler uses a HIWARE-proprietary object-file format when the `-Fh`, `-F6` or `-F7` options are set.

The HIWARE Object-file Format (`-Fh`) has the following limitations:

- The type char is limited to a size of 1 byte.
- Symbolic debugging for enumerations is limited to 16-bit signed enumerations.
- No zero bytes in strings are allowed (a zero byte marks the end of the string).

The HIWARE V2.7 Object-file Format (option `-F7`) has some limitations:

- The type char is limited to a size of 1 byte.
- Enumerations are limited to a size of 2 bytes and have to be signed.
- No symbolic debugging for enumerations.
- The standard type short is encoded as int in the object-file format.
- No zero bytes in strings allowed (a zero byte marks the end of the string).

The Compiler produces an ELF/DWARF object file when the `-F1` or `-F2` options are set. This object-file format may also be supported by other Compiler vendors.

In the Compiler ELF/DWARF 2.0 output, some constructs written in previous versions were not conforming to the ELF standard because the standard was not clear enough in this area. Because old versions of the simulator or debugger (V5.2 or earlier) are not able to load the corrected new format, the old behavior can still be produced by using "`-f2o`" instead of "`-f2`". Some old versions of the debugger (simulator or debugger V5.2 or earlier) generate a GPF when a new absolute file is loaded. If you want to use the older versions, use "`-f2o`" instead of "`-f2`". New versions of the debugger are able to load both formats correctly. Also, some older ELF/DWARF object file loaders from emulator vendors may require you to set the `-F2o` option.

The `-F1o` option is only supported if the target supports the ELF/DWARF 1.1 format. This option is only used with older debugger versions as a compatibility option. This option may be discontinued in the future. It is recommended you use `-F1` instead.

Note that it is recommended to use the ELF/DWARF 2.0 format instead of the ELF/DWARF 1.1. The 2.0 format is much more generic. In addition, it supports multiple include files plus modifications of the basic generic types (e.g., floating point format). Debug information is also more robust.

## **-Fd: Doubles are IEEE32**

### **Group**

CODE GENERATION

### **Scope**

Application

### **Syntax**

-Fd

### **Arguments**

None

### **Default**

None

### **Defines**

see -T: Flexible Type Management

### **Pragmas**

see -T

### **Description**

Allows you to change the float or double format. By default, float is IEEE32 and doubles are IEEE64.

With this option set, all doubles are in IEEE32 instead of IEEE64.

Floating point formats may be also changed with the -T option.

## **-H: Short Help**

### **Group**

VARIOUS

### **Scope**

None

### **Syntax**

-H

### **Arguments**

None

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

The -H option causes the Compiler to display a short list (i.e., help list) of available options within the Compiler window. Options are grouped into HOST, LANGUAGE, OPTIMIZATIONS, OUTPUT, INPUT, CODE GENERATION, MESSAGES, and VARIOUS.

No other option or source file should be specified when the -H option is invoked.

### **Example**

Listing 5.29 lists the short list options.

#### **Listing 5.29 Short Help options**

---

```
-H may produce the following list:  
INPUT:  
-!      Filenames are clipped to DOS length
```

## **Compiler Options**

*Compiler Option Details*

---

-I      Include file path  
VARIOUS:  
-H      Prints this list of options  
-V      Prints the Compiler version

---

## **-I: Include File Path**

### **Group**

INPUT

### **Scope**

Compilation Unit

### **Syntax**

`-I<path>`

### **Arguments**

`<path>`: path, terminated by a space or end-of-line

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

Allows you to set include paths in addition to the LIBPATH, LIBRARYPATH: 'include <File>' Path and GENPATH: #include "File" Path environment variables. Paths specified with this option have precedence over includes in the current directory, and paths specified in GENPATH, LIBPATH, and LIBRARYPATH.

### **Example**

---

```
-I. -I..\h -I\src\include
```

---

This directs the Compiler to search for header files first in the current directory (.), then relative from the current directory in ' . . \h ', and then in ' \src\include '. If the file is not found, the search continues with GENPATH, LIBPATH and LIBRARYPATH for header files in double quotes

## Compiler Options

### *Compiler Option Details*

---

(`#include "headerfile.h"`), and with `LIBPATH` and `LIBRARYPATH` for header files in angular brackets (`#include <stdio.h>`).

### **See also**

Input Files

-AddIncl: Additional Include File

LIBRARYPATH: 'include <File>' Path

GENPATH: #include "File" Path

## **-La: Generate Assembler Include File**

### **Group**

OUTPUT

### **Scope**

Function

### **Syntax**

`-La [= <filename>]`

### **Arguments**

`<filename>`: The name of the file to be generated  
It may contain special modifiers (see Using Special Modifiers)

### **Default**

No file created

### **Defines**

None

### **Pragmas**

None

### **Description**

The `-La` option causes the Compiler to generate an assembler include file when the `CREATE_ASM_LISTING` pragma occurs. The name of the created file is specified by this option. If no name is specified, a default of “%f.inc” is taken. To put the file into the directory specified by the `TEXTPATH: Text File Path` environment variable, use the option “`-la=%n.inc`”. The `%f` option already contains the path of the source file. When `%f` is used, the generated file is in the same directory as the source file.

The content of all modifiers refers to the main input file and not to the actual header file. The main input file is the one specified on the command line.

### **Example**

```
-La=asm.inc
```

## Compiler Options

*Compiler Option Details*

---

### See also

`#pragma CREATE_ASM_LISTING`: Create an Assembler Include File Listing

`-La`: Generate Assembler Include File



## **-Lasm: Generate Listing File**

### **Group**

OUTPUT

### **Scope**

Function

### **Syntax**

`-Lasm[=<filename>]`

### **Arguments**

<filename>: The name of the file to be generated.

It may contain special modifiers (see Using Special Modifiers).

### **Default**

No file created.

### **Defines**

None

### **Pragmas**

None

### **Description**

The `-Lasm` option causes the Compiler to generate an assembler listing file directly. All assembler generated instructions are also printed to this file. The name of the file is specified by this option. If no name is specified, a default of “%n.lst” is taken. The `TEXTPATH`: Text File Path environment variable is used if the resulting filename contains no path information.

The syntax does not always conform with the inline assembler or the assembler syntax. Therefore, this option can only be used to review the generated code. It can not currently be used to generate a file for assembly.

### **Example**

```
-Lasm=asm.lst
```

## **Compiler Options**

*Compiler Option Details*

---

### **See also**

-Lasmc: Configure Listing File

## **-Lasmc: Configure Listing File**

### **Group**

OUTPUT

### **Scope**

Function

### **Syntax**

```
-Lasmc [= {a | c | i | s | h | p | e | v | y} ]
```

### **Arguments**

a: Do not write the address in front of every instruction  
c: Do not write the hex bytes of the instructions  
i: Do not write the decoded instructions  
s: Do not write the source code  
h: Do not write the function header  
p: Do not write the source prolog  
e: Do not write the source epilog  
v: Do not write the compiler version  
y: Do not write cycle information

### **Default**

All printed together with the source

### **Defines**

None

### **Pragmas**

None

### **Description**

The `-Lasmc` option configures the output format of the listing file generated with the `-Lasm: Generate Listing File` option. The addresses, the hex bytes, and the instructions are selectively switched off.

The format of the listing file has layout shown in Listing 5.30. The letters in brackets ([ ]) indicate which suboption may be used to switch it off:

## Compiler Options

### Compiler Option Details

---

#### Listing 5.30 -Lasm configuration options

---

```
[v] ANSI-C/cC++ Compiler V-5.0.1
[v]
[p] 1:
[p] 2: void foo(void) {
[h]
[h] Function: foo
[h] Source : C:\Freescale\test.c
[h] Options : -Lasm=%n.lst
[h]
[s] 3: }
[a] 0000 [c] 3d [i] RTS
[e] 4:
[e] 5: // comments
[e] 6:
```

---

#### Example

```
-Lasmc=ac
```

## **-Ldf: Log Predefined Defines to File**

### **Group**

OUTPUT

### **Scope**

Compilation Unit

### **Syntax**

`-Ldf[=<file>]`

### **Arguments**

`<file>`: filename for the log file, default is `'predef.h'`.

### **Default**

default `<file>` is `'predef.h'`.

### **Defines**

None

### **Pragmas**

None

### **Description**

The `-Ldf` option causes the Compiler to generate a text file that contains a list of the compiler-defined `#define`. The default filename is `'predef.h'`, but may be changed (e.g., `-Ldf="myfile.h"`). The file is generated in the directory specified by the `TEXTPATH`: Text File Path environment variable. The defines written to this file depend on the actual Compiler option settings (e.g., type size settings, ANSI compliance, ...).

---

**NOTE** The defines specified by the command line (`-D`: Macro Definition option) are not included.

---

This option may be very useful for SQA. With this option it is possible to document every `#define` which was used to compile all sources.

## Compiler Options

### Compiler Option Details

---

**NOTE** This option only has an effect if a file is compiled. This option is unusable if you are not compiling a file.

---

### Example

Listing 5.31 is an example which lists the contents of a file containing define directives.

#### Listing 5.31 Displays the contents of a file where define directives are present

---

```
-Ldf
This generates the 'predef.h' filewith the following content:
/* resolved by preprocessor: __LINE__ */
/* resolved by preprocessor: __FILE__ */
/* resolved by preprocessor: __DATE__ */
/* resolved by preprocessor: __TIME__ */
#define __STDC__ 0
#define __VERSION__ 5004
#define __VERSION_STR__ "V-5.0.4"
#define __SMALL__
#define __PTR_SIZE_2__
#define __BITFIELD_LSBIT_FIRST__
#define __BITFIELD_MSBYTE_FIRST__
...
```

---

### See also

-D: Macro Definition

## -Li: List of Included Files

### Group

OUTPUT

### Scope

Compilation Unit

### Syntax

-Li

### Arguments

None

### Default

None

### Defines

None

### Pragmas

None

### Description

The `-Li` option causes the Compiler to generate a text file which contains a list of the `#include` files specified in the source. This text file shares the same name as the source file but with the extension, `*.inc`. The files are stored in the path specified by the `TEXTPATH: Text File Path` environment variable. The generated file may be used in make files.

### Example

Listing 5.32 is an example where the `-Li` compiler option can be used to display a file's contents when that file contains an included directive.

#### Listing 5.32 Display contents of a file when include directives are present

---

```
-Li  
If the source file is: 'C:\myFiles\b.c':
```

## Compiler Options

### Compiler Option Details

---

```
/* C:\myFiles\b.c */  
#include <string.h>
```

Then the generated file is:

```
C:\myFiles\b.c :\  
C:\Freescale\lib\targetc\include\string.h \  
C:\Freescale\lib\targetc\include\libdefs.h \  
C:\Freescale\lib\targetc\include\hidef.h \  
C:\Freescale\lib\targetc\include\stddef.h \  
C:\Freescale\lib\targetc\include\stdintypes.h
```

---

### See also

-Lm: List of Included Files in Make Format compiler option



## **-Lic: License Information**

### **Group**

VARIOUS

### **Scope**

None

### **Syntax**

`-Lic`

### **Arguments**

None

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

The `-Lic` option prints the current license information (e.g., if it is a demo version or a full version). This information is also displayed in the about box.

### **Example**

```
-Lic
```

### **See also**

#### **Compiler options:**

- `-LicA`: License Information about every Feature in Directory
- `-LicBorrow`: Borrow License Feature
- `-LicWait`: Wait until Floating License is Available from Floating License Server

## **-LicA: License Information about every Feature in Directory**

### **Group**

VARIOUS

### **Scope**

None

### **Syntax**

-LicA

### **Arguments**

None

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

The -LicA option prints the license information (e.g., if the tool or feature is a demo version or a full version) of every tool or \*.dll in the directory where the executable is located. This will take some time as every file in the directory is analyzed.

### **Example**

-LicA

### **See also**

#### **Compiler options:**

- -Lic: License Information
- -LicBorrow: Borrow License Feature
- -LicWait: Wait until Floating License is Available from Floating License Server

## **-LicBorrow: Borrow License Feature**

### **Group**

HOST

### **Scope**

None

### **Syntax**

```
-LicBorrow<feature>[;<version>]:<Date>
```

### **Arguments**

<feature>: the feature name to be borrowed (e.g., HI100100).

<version>: optional version of the feature to be borrowed (e.g., 3.000).

<date>: date with optional time until when the feature shall be borrowed (e.g., 15-Mar-2005:18:35).

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

This option allows to borrow a license feature until a given date or time. Borrowing allows you to use a floating license even if disconnected from the floating license server.

You need to specify the feature name and the date until you want to borrow the feature. If the feature you want to borrow is a feature belonging to the tool where you use this option, then you do not need to specify the version of the feature (because the tool knows the version). However, if you want to borrow any feature, you need to specify as well the feature version of it.

You can check the status of currently borrowed features in the tool about box.

## Compiler Options

### Compiler Option Details

---

---

**NOTE** You only can borrow features, if you have a floating license and if your floating license is enabled for borrowing. See as well the provided FLEXlm documentation about details on borrowing.

---

### Example

```
-LicBorrowHI100100;3.000:12-Mar-2006:18:25
```

### See also

#### Compiler options:

- -LicA: License Information about every Feature in Directory
- -Lic: License Information
- -LicWait: Wait until Floating License is Available from Floating License Server

## **-LicWait: Wait until Floating License is Available from Floating License Server**

### **Group**

HOST

### **Scope**

None

### **Syntax**

`-LicWait`

### **Arguments**

None

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

By default, if a license is not available from the floating license server, then the application will immediately return. With `-LicWait` set, the application will wait (blocking) until a license is available from the floating license server.

### **Example**

`-LicWait`

### **See also**

#### **Compiler options:**

- `-Lic`: License Information
- `-LicA`: License Information about every Feature in Directory
- `-LicBorrow`: Borrow License Feature

## **-Ll: Statistics about Each Function**

### **Group**

OUTPUT

### **Scope**

Compilation Unit

### **Syntax**

`-Ll [= <filename>]`

### **Arguments**

<filename>: file to be used for the output

### **Default**

The default output filename is `logfile.txt`

### **Defines**

None

### **Pragmas**

None

### **Description**

The `-Ll` option causes the Compiler to append statistical information about the compilation session to the specified file. Compiler options, code size (in bytes), stack usage (in bytes) and compilation time (in seconds) are given for each procedure of the compiled file. The information is appended to the specified filename (or the file `'make.txt'`, if no argument given). If the `TEXT_PATH: Text File Path` environment variable is set, the file is stored into the path specified by the environment variable. Otherwise it is stored in the current directory.

### **Example**

Listing 5.33 is an example where the use of the `-Ll` compiler options allows statistical information to be added to the end of an output listing file.

**Listing 5.33 Statistical information appended to assembler listing**

---

```
-Ll=mylog.txt
/* foo.c */
int Func1(int b) {
    int a = b+3;
    return a+2;
}
void Func2(void) {
}
Appends the following two lines into mylog.txt:
foo.c Func1 -Ll=mylog.txt    11  4 0.055000
foo.c Func2 -Ll=mylog.txt    1  0 0.001000
```

---

## **-Lm: List of Included Files in Make Format**

### **Group**

OUTPUT

### **Scope**

Compilation Unit

### **Syntax**

`-Lm[=<filename>]`

### **Arguments**

`<filename>`: file to be used for the output

### **Default**

The default filename is `Make.txt`

### **Defines**

None

### **Pragmas**

None

### **Description**

The `-Lm` option causes the Compiler to generate a text file which contains a list of the `#include` files specified in the source. The generated list is in a *make* format. The `-Lm` option is useful when creating make files. The output from several source files may be copied and grouped into one make file. The generated list is in the make format. The filename does not include the path. After each entry, an empty line is added. The information is appended to the specified filename (or the file 'make.txt', if no argument given). If the `TEXTPATH`: Text File Path environment variable is set, the file is stored into the path specified by the environment variable. Otherwise it is stored in the current directory.

### **Example**

Listing 5.34 is an example where the `-Lm` option generates a make file containing include directives.



**Listing 5.34 Make file construction**

---

```
COMPOTIONS=-Lm=mymake.txt
Compiling the following sources 'foo.c' and 'second.c':
/* foo.c */
#include <stddef.h>
#include "myheader.h"
...
/* second.c */
#include "inc.h"
#include "header.h"
...
This adds the following entries in the 'mymake.txt':
foo.o :      foo.c stddef.h myheader.h
seconde.o : second.c inc.h header.h
```

---

**See also**

- Li: List of Included Files
- Lo: Object File List
- Make Utility

## **-LmCfg: Configuration of list of Included files in Make Format**

### **Group**

OUTPUT

### **Scope**

Compilation Unit

### **Syntax**

`-LmCfg [= { i | l | m | o | u } ]`

### **Arguments**

i: Write path of included files  
l: Use line continuation  
m: Write path of main file  
o: Write path of object file  
u: Update information

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

This option is used when configuring the -Lm: List of Included Files in Make Format option. This option is operative only if the -Lm option is also used. The -Lm option produces the 'dependency' information for a make file. Each dependency information grouping is structured as shown:

```
<main object file>: <main source file> {<included  
file>}
```

### Example

If you compile a file named `b.c`, which includes `stdio.h`, the output of `-Lm` may be:

```
b.o: b.c stdio.h stddef.h stdarg.h string.h
```

The suboption, `'l'`, uses line continuation for each single entry in the dependency list. This improves readability as shown in Listing 5.35:

---

#### Listing 5.35 Using line continuations for clarity

---

```
b.o: \  
  b.c \  
  stdio.h \  
  stddef.h \  
  stdarg.h \  
  string.h
```

---

With the suboption `'m'`, the full path of the main file is written. The main file is the actual compilation unit (file to be compiled). This is necessary if there are files with the same name in different directories:

```
b.o: C:\test\b.c stdio.h stddef.h stdarg.h string.h
```

The suboption `'o'` has the same effect as `'m'`, but writes the full name of the target object file:

```
C:\test\obj\b.o: b.c stdio.h stddef.h stdarg.h string.h
```

The suboption `'i'` writes the full path of all included files in the dependency list:

```
b.o: b.c C:\Freescale\lib\include\stdio.h  
C:\Freescale\lib\include\stddef.h  
C:\Freescale\lib\include\stdarg.h  
C:\Freescale\lib\include\  
C:\Freescale\lib\include\string.h
```

The suboption `'u'` updates the information in the output file. If the file does not exist, the file is created. If the file exists and the current information is not yet in the file, the information is appended to the file. If the information is already present, it is updated. This allows you to specify this suboption for each compilation ensuring that the make dependency file is always up to date.

### Example

```
COMPOTIONS=-LmCfg=u
```

## Compiler Options

*Compiler Option Details*

---

### See also

- Li: List of Included Files
  - Lo: Object File List
  - Lm: List of Included Files in Make Format
- Make Utility

## **-Lo: Object File List**

### **Group**

OUTPUT

### **Scope**

Compilation Unit

### **Syntax**

`-Lo[=<filename>]`

### **Arguments**

`<filename>`: file to be used for the output

### **Default**

The default filename is `objlist.txt`

### **Defines**

None

### **Pragmas**

None

### **Description**

The `-Lo` option causes the Compiler to append the object filename to the list in the specified file. The information is appended to the specified filename (or the file 'make.txt', if no argument given). If `TEXTPATH: Text File Path` is set, the file is stored into the path specified by the environment variable. Otherwise, it is stored in the current directory.

### **See also**

`-Li`: List of Included Files  
`-Lm`: List of Included Files in Make Format

## **-Lp: Preprocessor Output**

### **Group**

OUTPUT

### **Scope**

Compilation Unit

### **Syntax**

`-Lp[=<filename>]`

### **Arguments**

`<filename>`: The name of the file to be generated.  
It may contain special modifiers (see Using Special Modifiers).

### **Default**

No file created

### **Defines**

None

### **Pragmas**

None

### **Description**

The `-Lp` option causes the Compiler to generate a text file which contains the preprocessor's output. If no filename is specified, the text file shares the same name as the source file but with the extension, `*.PRE(%n.pre)`. The `TEXTPATH` environment variable is used to store the preprocessor file.

The resultant file is a form of the source file. All preprocessor commands (i.e., `#include`, `#define`, `#ifdef`, etc.) have been resolved. Only source code is listed with line numbers.

### **See also**

- LpX: Stop after Preprocessor
- LpCfg: Preprocessor Output configuration

## **-LpCfg: Preprocessor Output configuration**

### **Group**

OUTPUT

### **Scope**

Compilation Unit

### **Syntax**

`-LpCfG [= { c | f | l | s } ]`

### **Arguments**

c: Do not generate line comments  
e: Generate empty lines  
f: Filenames with path  
l: Generate #line directives in preprocessor output  
m: Do not generate filenames  
s: Maintain spaces

### **Default**

If `-LpCfG` is specified, all suboptions (arguments) are enabled

### **Defines**

None

### **Pragmas**

None

### **Description**

The `-LpCfG` option specifies how source file and -line information is formatted in the preprocessor output. Switching `-LpCfG` off means that the output is formatted as in former compiler versions. The effects of the arguments are listed in Table 5.8.

**Table 5.8 Effects of Source and Line Information Format Control Arguments**

<b>Argument</b>	<b>on</b>	<b>off</b>
c	#line 1  #line 10	/* 1 */ /* 2 */ /* 10 */
e	int j;  int i;	int j;  int i;
f	C:\Freescale\include\stdlib.h	stdlib.h
l	#line 1 "stdlib.h"	**** FILE 'stdlib.h' */
m		**** FILE 'stdlib.h' */
s	/* 1 */ int f(void) { /* 2 */ return 1; /* 3 */ }	/* 1 */ int f ( void ) { /* 2 */ return 1 ; /* 3 */ }
all	#line 1 "C:\Freescale\include\stdlib.h"  #line 10	**** FILE 'stdlib.h' */ /* 1 */ /* 2 */ /* 10 */

**Example**

```
-Lpcfg
-Lpcfg=1fs
```

**See also**

```
-Lp: Preprocessor Output
```



## **-LpX: Stop after Preprocessor**

### **Group**

OUTPUT

### **Scope**

Compilation Unit

### **Syntax**

-LpX

### **Arguments**

None

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

Without this option, the compiler always translates the preprocessor output as C code. To do only preprocessing, use this option together with the `-Lp` option. No object file is generated.

### **Example**

-LpX

### **See also**

-Lp: Preprocessor Output

## **-M (-Ms, -Mt): Memory Model**

### **Group**

CODE GENERATION

### **Scope**

Application

### **Syntax**

-M(s | t)

### **Arguments**

s: small memory model

t: tiny memory model

### **Default**

-Ms

### **Defines**

\_\_SMALL\_\_

\_\_TINY\_\_

### **Pragmas**

None

### **Description**

See Memory Models

## **-N: Display Notify Box**

### **Group**

MESSAGES

### **Scope**

Function

### **Syntax**

-N

### **Arguments**

None

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

Makes the Compiler display an alert box if there was an error during compilation. This is useful when running a make file (please see *Make Utility*) because the Compiler waits for you to acknowledge the message, thus suspending make file processing. The 'N' stands for "Notify".

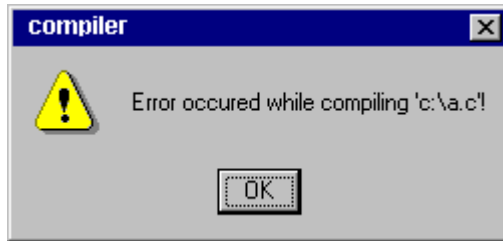
This feature is useful for halting and aborting a build using the Make Utility.

### **Example**

-N

If an error occurs during compilation, a dialog box (Figure 5.3) similar to the following one appears.

**Figure 5.3 Alert Dialog Box**



## **-NoBeep: No Beep in Case of an Error**

### **Group**

MESSAGES

### **Scope**

Function

### **Syntax**

-NoBeep

### **Arguments**

None

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

There is a 'beep' notification at the end of processing if an error was generated. To implement a silent error, this 'beep' may be switched off using this option.

### **Example**

-NoBeep

## **-NoDebugInfo: Do not Generate Debug Information**

### **Group**

OUTPUT

### **Scope**

None

### **Syntax**

-NoDebugInfo

### **Arguments**

None

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

The compiler generates debug information by default. When this option is used, the compiler does not generate debug information.

---

**NOTE** To generate an application without debug information in ELF, the linker provides an option to strip the debug information. By calling the linker twice, you can generate two versions of the application: one with and one without debug information. This compiler option has to be used only if object files or libraries are to be distributed without debug info.

---

---

**NOTE** This option does not affect the generated code. Only the debug information is excluded.

---

**See also**

**Compiler options:**

-F (-Fh, -F1, -F1o, -F2, -F2o, -F6, or -F7): Object-File Format

-NoPath: Strip Path Info

## **-NoEnv: Do not Use Environment**

### **Group**

STARTUP. This option cannot be specified interactively.

### **Scope**

None

### **Syntax**

-NoEnv

### **Arguments**

None

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

This option can only be specified at the command line while starting the application. It can not be specified in any other way, including via the `default.env` file, the command line, or processes.

When this option is given, the application does not use any environment (`default.env`, `project.ini`, or tips file) data.

### **Example**

```
compiler.exe -NoEnv
```

Use the compiler executable name instead of “compiler”.

### **See also**

Local Configuration File (usually `project.ini`)



## **-NoPath: Strip Path Info**

### **Group**

OUTPUT

### **Scope**

Compilation Unit

### **Syntax**

-NoPath

### **Arguments**

None

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

With this option set, it is possible to avoid any path information in object files. This is useful if you want to move object files to another file location, or to hide your path structure.

### **See also**

-NoDebugInfo: Do not Generate Debug Information

## **-O (-Os, -Ot): Main Optimization Target**

### **Group**

OPTIMIZATIONS

### **Scope**

Function

### **Syntax**

`-O(s|t)`

### **Arguments**

s: Optimization for code size (default)

t: Optimization for execution speed

### **Default**

`-Os`

### **Defines**

`__OPTIMIZE_FOR_SIZE__`

`__OPTIMIZE_FOR_TIME__`

### **Pragmas**

None

### **Description**

There are various points where the Compiler has to choose between two possibilities: it can either generate fast, but large code, or small but slower code.

The Compiler generally optimizes on code size. It often has to decide between a runtime routine or an expanded code. The programmer can decide whether to choose between the slower and shorter or the faster and longer code sequence by setting a command line switch.

The `-Os` option directs the Compiler to optimize the code for smaller code size. The Compiler trades faster-larger code for slower-smaller code.

The `-Ot` option directs the Compiler to optimize the code for faster execution time. The Compiler will “trade” slower-smaller code for faster-larger code.

---

**NOTE** This option only affects some special code sequences. This option has to be set together with other optimization options (e.g., register optimization) to get best results.

---

**Example**

`-Os`

## **-Obfv: Optimize Bitfields and Volatile Bitfields**

### **Group**

OPTIMIZATIONS

### **Scope**

Function

### **Syntax**

-Obfv

### **Arguments**

None

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

Optimize bitfields as well as bitfields declared as volatile. The compiler is allowed to change the access order or to combine many accesses to one, even if the bitfields are declared as volatile.

### **Example**

Listing 5.36 contains bitfields to be optimized with the -Obfv compiler option.

#### **Listing 5.36 Bitfields example**

---

```
volatile struct {  
    unsigned int b0:1;  
    unsigned int b1:1;  
    unsigned int b2:1;  
} bf;  
void foo(void) {
```

```
    bf.b0 = 1;  bf.b1 = 1;  bf.b2 = 1;  
}
```

---

using -Obfv:

```
BSET  bf, #7
```

without -Obfv:

```
BSET  bf, #1
```

```
BSET  bf, #2
```

```
BSET  bf, #4
```

## **-ObjN: Object filename Specification**

### **Group**

OUTPUT

### **Scope**

Compilation Unit

### **Syntax**

-ObjN=<file>

### **Arguments**

<file>: Object filename

### **Default**

-ObjN=% (OBJPATH) \ %n . o

### **Defines**

None

### **Pragmas**

None

### **Description**

The object file has the same name as the processed source file, but with the extension “\*.o”. This option allows a flexible way to define the object filename. It may contain special modifiers (see Using Special Modifiers). If <file> in the option contains a path (absolute or relative), the OBJPATH environment variable is ignored.

### **Listing 5.37 Example**

---

```
-ObjN=a.out
```

---

The resulting object file is “a.out”. If the OBJPATH environment variable is set to “\src\obj”, the object file is “\src\obj\a.out”.

```
fibonacci.c -ObjN=%n.obj
```

The resulting object file is “fibo.obj”.

```
myfile.c -ObjN=..\objects\_%n.obj
```

The object file is named relative to the current directory to

“..\objects\\_myfile.obj”. The OBJPATH environment variable is ignored as the <file> contains a path.

**See also**

OBJPATH: Object File Path environment variable

## **-Oc: Common Subexpression Elimination (CSE)**

### **Group**

OPTIMIZATIONS

### **Scope**

Function

### **Syntax**

-Oc

### **Arguments**

None

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

Performs common subexpression elimination (CSE). The code for common subexpressions and assignments is generated only once. The result is reused. Depending on available registers, a common subexpression may produce more code due to many spills.

---

**NOTE** When the CSE is switched on, changes of variables by aliases may generate incorrect optimizations.

---

This option is disabled and present only for compatibility reasons for the Freescale HC(S)08

### **Example**

-Oc



Listing 5.38 is an example where the use of the CSE compiler option causes incorrect optimizations. But, no matter, because this option is not enabled any longer in any event for the HC(S)08.

**Listing 5.38 Example where CSE may produce incorrect results**

---

```
void main(void) {
    int x;
    int *p;
    x = 7; /* here the value of x is set to 7 */
    p = &x;
    *p = 6; /* here x is set to 6 by the alias *p */
    /* here x is assumed to be equal to 7 and
       Error is called */
    if(x != 6) Error();
}
```

---

**NOTE** This error does not occur if x is declared as volatile.

---

## **-OdocF: Dynamic Option Configuration for Functions**

### **Group**

OPTIMIZATIONS

### **Scope**

Function

### **Syntax**

`-OdocF="<option>"`

### **Arguments**

*<option>*: Set of options, separated by `|` to be evaluated for each single function.

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

Normally, you must set a specific set of Compiler switches for each compilation unit (file to be compiled). For some files, a specific set of options may decrease the code size, but for other files, the same set of Compiler options may produce more code depending on the sources.

Some optimizations may reduce the code size for some functions, but may increase the code size for other functions in the same compilation unit. Normally it is impossible to vary options over different functions, or to find the best combination of options.

This option solves this problem by allowing the Compiler to choose from a set of options to reach the smallest code size for every function. Without this feature, you must set some Compiler switches, which are fixed, over the whole compilation unit. With this feature, the Compiler is free to find the best option combination from a user-defined set for every function.

Standard merging rules applies also for this new option, e.g.,

```
-Or -OdocF="-Ocu|-Cu"
```

is the same as:

```
-OrDOCF="-Ouc|-Cu"
```

The Compiler attempts to find the best option combination (of those specified) and evaluates all possible combinations of all specified sets, e.g., for the option shown in Listing 5.39:

---

**Listing 5.39 Example of dynamic option configuration**

---

```
-W2 -OdocF="-Or|-Cni|-Cu|-Oc"
```

---

The code sizes for following option combinations are evaluated:

1. -W2
2. -W2 -Or
3. -W2 -Cni -Cu
4. -W2 -Or -Cni -Cu
5. -W2 -Oc
6. -W2 -Or -Oc
7. -W2 -Cni -Cu -Oc
8. -W2 -Or -Cni -Cu -Oc

Thus, if the more sets are specified, the longer the Compiler has to evaluate all combinations, e.g., for 5 sets 32 evaluations.

---

**NOTE** No options with scope Application or Compilation Unit (as memory model, float or double format, or object-file format) or options for the whole compilation unit (like inlining or macro definition) should be specified in this option. The generated functions may be incompatible for linking and executing.

---

**Limitations:**

- The maximum set of options set is limited to five, e.g.,  
-OdocF="-Or -Ou|-Cni|-Cu|-Oic2|-W2 -Ob"
- The maximum length of the option is 64 characters.
- The feature is available only for functions and options compatible with functions. Future extensions will also provide this option for compilation units.

**Example**

```
-Odocf="-Or|-Cni"
```

## **-Of and-Onf: Create Sub-Functions with Common Code**

### **Group**

OPTIMIZATIONS

### **Scope**

Function

### **Syntax**

-Onf

### **Arguments**

None

### **Default**

-Of default or with -Os, -Onf with -Ot

### **Defines**

None

### **Pragmas**

None

### **Description**

This option performs the reverse of inlining. It detects common code parts in the generated code. The common code is moved to a different place and all occurrences are replaced with a JSR to the moved code. At the end of the common code, an RTS instruction is inserted. All SP uses are increased by an address size. This optimization takes care of stack allocation, control flow, and of functions having arguments on the stack. Also, inline assembler code is never treated as common code.

### **Example**

Consider the function in Listing 5.40:

#### **Listing 5.40 Example function**

---

```
void f(int);  
void g(void);  
void h(void);
```

```
void main(void) {
    f(1); f(2); f(3);
    h();
    f(1); f(2); f(3);
    g();
    f(1); f(2);
}
```

---

The compiler first detects that "f(1); f(2); f(3);" occurs twice and places this code separately. The two code patterns are replaced by a call to the moved code.

This situation can be thought of as the following non-C pseudo code (C does not support local functions) (Listing 5.41):

#### **Listing 5.41 local tmp0() function**

---

```
void main(void) {
    void tmp0(void) {
        f(1); f(2); f(3);
    }
    tmp0();
    h();
    tmp0();
    g();
    f(1); f(2);
}
```

---

In a next step, the compiler detects that the code "f(1); f(2);" occurs twice. The Compiler generates a second internal function (Listing 5.42):

#### **Listing 5.42 another local function - tmp1()**

---

```
void main(void) {
    void tmp1(void) {
        f(1); f(2);
    }
    void tmp0(void) {
        tmp1(); f(3);
    }
    tmp0();
    h();
    tmp0();
    g();
}
```

## Compiler Options

### Compiler Option Details

---

```
    tmp1 ();  
}
```

---

The new code of the function `tmp1` (actually `tmp1` is not really a function - it is a part of the `main()` function) is called once directly from `main()`, and once indirectly by using `tmp0`. These two call chains now use a different amount of stack. Because of this situation, it is not always possible to generate correct debug information. For the local function `tmp1`, the compiler cannot state both possible SP values. It only states one of them. While debugging the other state, local variables and the call chain are declared invalid in the debugger.

---

**TIP** Switch off this optimization to debug your application. The common code makes the control flow more complicated. Also the debugger cannot distinguish between two distinct usages of the same code. Setting a breakpoint in common code stops the application and every use of it. It will also stop the local variables and the call frame if not displayed correctly, as explained above.

---

---

**TIP** Switch off this optimization to get faster code. For code density, there are only a few cases where the code gets worse. This situation only occurs when other optimizations (such as branch tail merging or peepholing) do not find a pattern.

---

## **-Oi: Inlining**

### **Group**

OPTIMIZATIONS

### **Scope**

Compilation unit

### **Syntax**

`-Oi [= (c<code Size> | OFF) ]`

### **Arguments**

`<code Size>`: Limit for inlining in code size

`OFF`: switching off inlining

### **Default**

None. If no `<code Size>` is specified, the compiler uses a default code size of 40 bytes

### **Defines**

None

### **Pragmas**

`#pragma INLINE`

### **Description**

This option enables inline expansion. If there is a `#pragma INLINE` before a function definition, all calls of this function are replaced by the code of this function, if possible.

Using the option `-Oi=c0` switches off inlining. Functions marked with the `#pragma INLINE` are still inlined. To disable inlining, use the `-Oi=OFF` option.

### **Example**

---

```
-Oi
#pragma INLINE
static void f(int i) {
    /* all calls of function f() are inlined */
```

## Compiler Options

### Compiler Option Details

---

```
/* ... */  
}
```

---

The [=c<n>] option extension signifies that all functions with a size smaller than <n> are inlined. For example, compiling with the option `-oi=c100` enables inline expansion for all functions with a size smaller than 100 bytes.

### Restrictions

The following functions are not inlined:

- functions with default arguments
- functions with labels inside
- functions with an open parameter list (“void f(int i, ...);”)
- functions with inline assembly statements
- functions using local static objects



## **-Oilib: Optimize Library Functions**

### **Group**

OPTIMIZATIONS

### **Scope**

Function

### **Syntax**

`-Oilib[=<arguments>]`

### **Arguments**

`<arguments>` are one or multiple of following suboptions:

- a: inline calls to `strcpy()`
- b: inline calls to `strlen()`
- d: inline calls to `fabs()` or `fabsf()`
- e: inline calls to `memset()`
- f: inline calls to `memcpy()`
- g: replace shifts left of 1 by array lookup

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

This option enables the compiler to optimize specific known library functions to reduce execution time. The Compiler frequently uses small functions such as `strcpy()`, `strcmp()`, and so forth. The following functions are optimized:

- `strcpy()` (only available for ICG-based backends)
- `strlen()` (e.g., `strlen("abc")`)

## Compiler Options

### Compiler Option Details

---

memset() or memcpy() (pg. 2 - Table 01.1)

- `memset()`

`memset()` is optimized only if:

- the result is not used
- `memset()` is used to zero out
- the size for the zero out is in the range 1 to 0xff
- the ANSI library header file `<string.h>` is included

An example for this is `(void)memset(&buf, 0, 50);` In this case, the call to `memset()` is replaced with a call to `'_memset_clear_8bitCount'` present in the ANSI library (`string.c`)

`memcpy()` is optimized only if:

- the result is not used,
- the size for the copy out is in the range 0 – 0xff,
- the ANSI library header file `<string.h>` is included.

An example for this is `(void)memcpy(&buf, &buf2, 30);`

In this case the call to `memcpy()` is replaced with a call to `'_memcpy_8bitCount'` present in the ANSI library (`string.c`)

`(char)1 << val` is replaced by `_PowOfTwo_8[val]` if `_PowOfTwo_8` is known at compile time. Similarly, for 16-bit and for 32-bit shifts, the arrays `_PowOfTwo_16` and `_PowOfTwo_32` are used. These constant arrays contain the values 1, 2, 4, 8... They are declared in `hedef.h`. This optimization is performed only when optimizing for time.

`-Oilib` without arguments: optimize calls to all supported library functions.

### Example

Compiling the function `f` below with the `-Oilib=a` option (only available for ICG-based backends)

```
void f(void) {
    char *s = strcpy(s, ct);
}
```

is translated similar to the following function:

```
void g(void) {
    s2 = s;
    while(*s2++ = *ct++);
}
```

**See also**

-Oi: Inlining  
Message C5920

## **-O1: Try to Keep Loop Induction Variables in Registers**

### **Group**

OPTIMIZATIONS

### **Scope**

Function

### **Syntax**

`-O1<number>`

### **Arguments**

`<number>`: number of registers to be used for induction variables

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

Try to maintain `<number>` loop induction variables in registers. Loop induction variables are variables read and written within the loop (e.g., loop counter). The Compiler tries to keep loop induction variables in registers to reduce execution time, and sometimes also code size. This option sets the number of loop induction variables the Compiler is allowed to keep in registers. The range is from 0 (no variable) to infinity. If this option is not given, the Compiler takes the optimal number (code density). Like the option `-or`, this option may increase code size (spill and merge code) if too many loop induction variables are specified.

---

**NOTE** Disable this option (with `-O10`) if there are problems when debugging your code. This optimization could increase the complexity of code debugging on a High-Level Language level.

---

The example in Listing 5.43 is used in Listing 5.44 and in Listing 5.45.

**Listing 5.43 Example (abstract code)**

---

```
void main(char *s) {
    do {
        *s = 0;
    } while (*++s);
}
```

---

Listing 5.44 shows pseudo disassembly with the `-O10` option:

**Listing 5.44 With the `-O10` option (no optimization, pseudo code)**

---

```
loop:
    LD  s, Reg0
    ST  #0, [Reg0]
    INC Reg0
    ST  Reg0, s
    CMP [Reg0], #0
    BNE loop
```

---

Listing 5.45 shows pseudo disassembly without the `-O1` option (i.e., optimized) where the load and stores from or to variable `s` disappear.

**Listing 5.45 Without option (optimized, pseudo assembler)**

---

```
loop:
    ST  #0, s
    INC s
    CMP s, #0
    BNE loop
```

---

**Example**

`-O11`

## **-Ona: Disable Alias Checking**

### **Group**

OPTIMIZATIONS

### **Scope**

Function

### **Syntax**

-Ona

### **Arguments**

None

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

Variables that may be written by a pointer indirection or an array access are redefined after the optimization. This option prevents the Compiler from doing this redefinition, which may allow you to reuse already-loaded variables or equivalent constants. Use this option only if you are sure you will have no real writes of aliases to a memory location of a variable.

### **Example**

do not compile with -Ona.

```
void main(void) {
    int a = 0, *p = &a;
    *p = 1; // real write by the alias *p
    if(a == 0) Error(); // if -Ona is specified,
    // Error() is called!
}
```

## **-OnB: Disable Branch Optimizer**

### **Group**

OPTIMIZATIONS

### **Scope**

Function

### **Syntax**

`-OnB[=<option Char>{<option Char>}]`

### **Arguments**

<option Char> is one of the following:

- a: Disable short BRA optimization
- b: Disable Branch JSR to BSR optimization
- c: Disable branch to branch optimization
- d: Disable dead code optimization
- l: Disable long branch optimization
- r: Disable Branch to RTS optimization
- t: Disable Branch tail optimization

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

See Backend for details

### **Example**

`-OnB`

Disables all branch optimizations

**See also**

Branch Optimizations



## **-Onbf: Disable Optimize Bitfields**

### **Group**

OPTIMIZATIONS

### **Scope**

Function

### **Syntax**

-Onbf

### **Arguments**

None

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

A sequence of bitfield assignments with constants is not combined if you use -Onbf. This option simplifies debugging and makes the code more readable.

### **Example**

#### **Listing 5.46 Example bitfield definition**

---

```
struct {
    b0:1;
    b1:1;
    b2:1;
} bf;

void main(void) {
    bf.b0 = 0;
}
```

## Compiler Options

### Compiler Option Details

---

```
bf.b1 = 0;  
bf.b2 = 0;  
}
```

---

without `-Onbf`: (pseudo intermediate code)

```
BITCLR bf, #7 // all 3 bits (the mask is 7)  
           // are cleared
```

with `-Onbf`: (pseudo intermediate code)

```
BITCLR bf, #1 // clear bit 1 (mask 1)  
BITCLR bf, #2 // clear bit 2 (mask 2)  
BITCLR bf, #4 // clear bit 3 (mask 4)
```

### Example

`-Onbf`

## **-Onbt: Disable ICG Level Branch Tail Merging**

### **Group**

OPTIMIZATIONS

### **Scope**

Function

### **Syntax**

-Onbt

### **Arguments**

None

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

The ICG level branch tail merging is switched off leading to more readable code and simplified debugging.

The example in Listing 5.47 disassembles to the following pseudocode.

#### **Listing 5.47 Example function**

---

```
void main(void) {  
  
    if(x > 0) {  
        y = 4;  
    } else {  
        y = 9;  
    }  
}
```

---

## Compiler Options

### Compiler Option Details

---

Without `-Onbt`, the above example disassembles as in Listing 5.48

#### Listing 5.48 Case (1) without `-Onbt`: (pseudo intermediate code)

---

```
        CMP    x, 0
        BLE    else_label

        LOAD   reg, #4
        BRA    branch_tail

else_label: LOAD   reg, #9
branch_tail: STORE y, reg
go_on:     ...
```

---

With the `-Obnt` compiler option, Listing 5.47 disassembles as in Listing 5.49.

#### Listing 5.49 Case (2) with `-Onbt`: (pseudo intermediate code)

---

```
        CMP    x, 0
        BLE    else_label

        LOAD   reg, #4
        STORE  y, reg
        BRA    go_on

else_label: LOAD   reg, #9
           STORE  y, reg
go_on:     ...
```

---

## Example

`-Onbt`

## -Onca: Disable any Constant Folding

### Group

OPTIMIZATIONS

### Scope

Function

### Syntax

-Onca

### Arguments

None

### Default

None

### Defines

None

### Pragmas

None

### Description

Disables any constant folding over statement boundaries. This option prevents the Compiler from folding constants. All arithmetical operations are coded. This option must be set when the library functions, `setjmp()` and `longjmp()`, are present. If this option is not set, the Compiler makes wrong assumptions as in the example in Listing 5.50:

#### Listing 5.50 Example with if condition always true

---

```
void main(void) {
    jmp_buf env;
    int k = 0;
    if (setjmp(env) == 0) {
        k = 1;
        longjmp(env, 0);
        Err(1);
    }
}
```

## Compiler Options

### *Compiler Option Details*

---

```
    } else if (k != 1) { /* assumed always TRUE */  
        Err(0);  
    }  
}
```

---

### **Example**

-Onca

## **-Oncn: Disable Constant Folding in case of a New Constant**

### **Group**

OPTIMIZATIONS

### **Scope**

Function

### **Syntax**

-Oncn

### **Arguments**

None

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

Disables any constant folding in the case of a new constant. This option prevents the Compiler from folding constants if the resulting constant is new.

The option only has an effect for processors where a constant is difficult to load (e.g., RISC processors).

### **Listing 5.51 Example (pseudo code)**

---

```
void main(void) {  
    int a = 1, b = 2, c, d;  
  
    c = a + b;  
    d = a * b;  
}
```

---

## Compiler Options

### Compiler Option Details

---

Case (1) without the `-Oncn` option (pseudo code):

```
a MOVE 1
b MOVE 2
c MOVE 3
d MOVE 2
```

Case (2) with the `-Oncn` option (pseudo code):

```
a MOVE 1
b MOVE 2
c ADD a, b
d MOVE 2
```

The constant 3 is a new constant that does not appear in the source. The constant 2 is already present, so it is still propagated.

### Example

```
-Oncn
```



## **-OnCopyDown: Do Generate Copy Down Information for Zero Values**

### **Group**

OPTIMIZATIONS

### **Scope**

Compilation unit

### **Syntax**

-OnCopyDown

### **Arguments**

None

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

With usual startup code, all global variables are first set to 0 (zero out). If the definition contained an initialization value, this initialization value is copied to the variable (copy down). Because of this, it is not necessary to copy zero values unless the usual startup code is modified. If a modified startup code contains a copy down but not a zero out, use this option to prevent the compiler from removing the initialization.

---

**NOTE** The case of a copy down without a zero out is normally not used. Because the copy down needs much more space than the zero out, it usually contains copy down and zero out, zero out alone, or none of them.

---

## Compiler Options

### Compiler Option Details

---

In the HIWARE format, the object-file format permits the Compiler to remove single assignments in a structure or array initialization. In the ELF format, it is optimized only if the whole array or structure is initialized with 0.

---

**NOTE** This option controls the optimizations done in the compiler. However, the linker itself might further optimize the copy down or the zero out.

---

### Example

```
int i=0;
int arr[10]={1,0,0,0,0,0,0,0,0,0};
```

If this option is present, no copy down is generated for `i`. For the `arr` array, the initialization with 0 can only be optimized in the HIWARE format. In ELF it is not possible to separate them from the initialization with 1.

## -OnCstVar: Disable CONST Variable by Constant Replacement

### Group

OPTIMIZATIONS

### Scope

Compilation Unit

### Syntax

-OnCstVar

### Arguments

None

### Default

None

### Defines

None

### Pragmas

None

### Description

This option provides you with a way to switch OFF the replacement of CONST variable by the constant value.

### Example

```
const int MyConst = 5;
int i;
void foo(void) {
    i = MyConst;
}
```

If the `-OnStVar` option is not set, the compiler replaces each occurrence of 'MyConst' with its constant value 5; that is 'i = MyConst' will be transformed into 'i = 5;'. The Memory or ROM needed for the 'MyConst' constant variable is optimized as well. With the `-OnCstVar` option set, this optimization is avoided. This is logical only if you want to have unoptimized code.

## **-One: Disable any Low-level Common Subexpression Elimination**

### **Group**

OPTIMIZATIONS

### **Scope**

Function

### **Syntax**

-One

### **Arguments**

None

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

This option prevents the Compiler from reusing common subexpressions, such as array indexes and array base addresses. The code size may increase. The low-level CSE does not have the alias problems of the frontend CSE and is therefore switched on by default.

The two CSE optimizations do not cover the same cases. The low-level CSE has a finer granularity but does not handle all cases of the frontend CSE.

Use this option only to generate more readable code for debugging.

### **Listing 5.52 Example (abstract code)**

---

```
void main(int i) {  
    int a[10];
```

```
    a[i] = a[i-1];  
}
```

---

See Listing 5.53 for case not using `-One` and Listing 5.54 for a case not using `-One`.

**Listing 5.53 Case (1) without option (optimized)**

---

```
tmp1    LD    i  
tmp2    LSL   tmp1, #1  
tmp3    SUB   tmp2, #2  
tmp4    ADR   a  
tmp5    ADD   tmp3, tmp4  
tmp6    LD    (tmp5)  
2 (tmp5) ST   tmp6
```

---

**Listing 5.54 Case (2) `-One` (not optimized, readable)**

---

```
tmp1    LD    i  
tmp2    LSL   tmp1, #1  
tmp3    SUB   tmp2, #2  
tmp4    ADR   a  
tmp5    ADD   tmp3, tmp4  
tmp6    LSL   tmp1, #1    ;calculated twice  
tmp7    ADR   a            ;calculated twice  
tmp8    ADD   tmp6, tmp7  
tmp9    LD    (tmp5)  
(tmp8) ST   tmp9
```

---

**Example**

`-One`

## **-OnP: Disable Peephole Optimization**

### **Group**

OPTIMIZATIONS

### **Scope**

Function

### **Syntax**

`-OnP[=<option Char>{<option Char>}]`

### **Arguments**

<option Char> is one of the following:

- a: Disable peephole combine AI (S|X) optimization
- b: Disable peephole handle constant argument optimization
- c: Disable peephole PSH/PUL instead AIS optimization
- d: Disable peephole combine bit operations optimization
- e: Disable peephole combine bit set clr optimization
- f: Disable peephole PSH PUL optimization
- g: Disable peephole RTS RTS optimization
- h: Disable peephole unused loads optimization
- i: Disable peephole unused stores optimization
- j: Disable peephole unused compares optimization
- k: Disable peephole unnecessary tests optimization
- l: Disable peephole unnecessary transfers optimization
- m: Disable peephole JSR to JMP optimization
- n: Disable peephole CMP #1 optimization
- o: Disable peephole simple inline assembler optimizations

### **Default**

None

### **Defines**

None

## Pragmas

None

## Description

If `-OnP` is specified, the whole peephole optimizer is disabled. To disable only a single peephole optimization, the optional syntax `-OnP=<char>` may be used. For example:

Suboption `-OnP=m` (Disable peephole JSR to JMP optimization)

The compiler replaces a JSR- RTS sequence with a single JMP instruction. This saves single-byte code and 2-byte stack spaces. To avoid this optimization, use the `-onp=m` option.

### Code Example:

with `-onp=m`:

```
...  
JSR   Subroutine  
RTS
```

without `-onp=m`:

```
...  
JMP   Subroutine
```

## Example

`-OnP=m`

Disables the peephole optimization 'm'

## **-OnPMNC: Disable Code Generation for NULL Pointer to Member Check**

### **Group**

OPTIMIZATIONS

### **Scope**

Compilation Unit

### **Syntax**

-OnPMNC

### **Arguments**

None

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

Before assigning a pointer to a member in C++, you must ensure that the pointer to the member is not NULL in order to generate correct and safe code. In embedded systems development, the problem is to generate the denser code while avoiding overhead whenever possible (this NULL check code is a good example). If you can ensure this pointer to a member will never be NULL, then this NULL check is useless. This option enables you to switch off the code generation for the NULL check.

### **Example**

-OnPMNC



## -Ont: Disable Tree Optimizer

### Group

OPTIMIZATIONS

### Scope

Function

### Syntax

```
-Ont [= { % | & | * | + | - | / | 0 | 1 | 7 | 8 | 9 | ? | ^ | a | b | c | d | e |  
f | h | i | l | m | n | o | p | q | r | s | t | u | v | w | | | ~ } ]
```

### Arguments

- %: Disable mod optimization
- &: Disable bit and optimization
- \*: Disable mul optimization
- +: Disable plus optimization
- : Disable minus optimization
- /: Disable div optimization
- 0: Disable and optimization
- 1: Disable or optimization
- 7: Disable extend optimization
- 8: Disable switch optimization
- 9: Disable assign optimization
- ?: Disable test optimization
- ^: Disable xor optimization
- a: Disable statement optimization
- b: Disable constant folding optimization
- c: Disable compare optimization
- d: Disable binary operation optimization
- e: Disable constant swap optimization
- f: Disable condition optimization
- h: Disable unary minus optimization
- i: Disable address optimization
- j: Disable transformations for inlining
- l: Disable label optimization
- m: Disable left shift optimization
- n: Disable right shift optimization
- o: Disable cast optimization

## Compiler Options

### Compiler Option Details

---

p: Disable cut optimization  
q: Disable 16-32 compare optimization  
r: Disable 16-32 relative optimization  
s: Disable indirect optimization  
t: Disable for optimization  
u: Disable while optimization  
v: Disable do optimization  
w: Disable if optimization  
|: Disable bit or optimization  
~: Disable bit neg optimization

#### Default

If `-Ont` is specified, all optimizations are disabled

#### Defines

None

#### Pragmas

None

#### Description

The Compiler contains a special optimizer which optimizes the internal tree data structure. This tree data structure holds the semantic of the program and represents the parsed statements and expressions.

This option disables the tree optimizer. This may be useful for debugging and for forcing the Compiler to produce ‘straightforward’ code. Note that the optimizations below are just examples for the classes of optimizations.

If this option is set, the Compiler will not perform the following optimizations:

#### **-Ont=~**

Disable optimization of ‘`~i`’ into ‘`i`’

#### **-Ont=|**

Disable optimization of ‘`i | 0xffff`’ into ‘`0xffff`’

#### **-Ont=w**

Disable optimization of ‘`if (1) i = 0;`’ into ‘`i = 0;`’

**-Ont=v**

Disable optimization of ‘do ... while(0) into ‘...’

**-Ont=u**

Disable optimization of ‘while(1) ...;’ into ‘...;’

**-Ont=t**

Disable optimization of ‘for(;;) ...’ into ‘while(1) ...’

**-Ont=s**

Disable optimization of ‘\*&i’ into ‘i’

**-Ont=r**

Disable optimization of ‘L<=4’ into 16-bit compares if 16-bit compares are better

**-Ont=q**

Reduction of long compares into int compares if int compares are better: (-Ont=q to disable it)

```
if (uL == 0)
```

will be optimized into

```
if ((int)(uL>>16) == 0 && (int)uL == 0)
```

**-Ont=p**

Disable optimization of ‘(char)(long)i’ into ‘(char)i’

**-Ont=o**

Disable optimization of ‘(short)(int)L’ into ‘(short)L’ if short and int have the same size

**-Ont=n, -Ont=m:**

Optimization of shift optimizations (<<, >>, -Ont=n or -Ont=m to disable it):  
Reduction of shift counts to unsigned char:

```
uL = uL1 >> uL2;
```

will be optimized into

```
uL = uL1 >> (unsigned char)uL2;
```

Optimization of zero shift counts:

```
uL = uL1 >> 0;
```

will be optimized into

```
uL = uL1;
```

Optimization of shift counts greater than the object to be shifted:

```
uL = uL1 >> 40;
```

will be optimized into

```
uL = 0L;
```

Strength reduction for operations followed by a cast operation:

```
ch = uL1 * uL2;
```

will be optimized into

```
ch = (char)uL1 * (char)uL2;
```

Replacing shift operations by load or store

```
i = uL >> 16;
```

will be optimized into

```
i = *(int *)(&uL);
```

Shift count reductions:

```
ch = uL >> 17;
```

will be optimized into

```
ch = (*(char *)(&uL)+1)>>1;
```

Optimization of shift combined with binary and:

```
ch = (uL >> 25) & 0x10;
```

will be optimized into

```
ch = ((* (char *) (&uL)) >> 1) & 0x10;
```

**-Ont=l**

Disable optimization removal of labels if not used

**-Ont=i**

Disable optimization of ‘&\*p’ into ‘p’

**-Ont=j**

This optimization does transform the syntax tree into an equivalent form in which more inlining cases can be done. This option only has an effect when inlining is enabled.

**-Ont=h**

Disable optimization of ‘-(-i)’ into ‘i’

**-Ont=f**

Disable optimization of ‘(a==0)’ into ‘(!a)’

**-Ont=e**

Disable optimization of ‘2\*i’ into ‘i\*2’

**-Ont=d**

Disable optimization of ‘us & ui’ into ‘us & (unsigned short)ui’

**-Ont=c**

Disable optimization of ‘if ((long)i)’ into ‘if (i)’

**-Ont=b**

Disable optimization of ‘3+7’ into ‘10’

**-Ont=a**

Disable optimization of last statement in function if result is not used

## Compiler Options

### Compiler Option Details

---

#### **-Ont=^**

Disable optimization of 'i^0' into 'i'

#### **-Ont=?**

Disable optimization of 'i = (int) (cond ? L1:L2);' into  
'i = cond ? (int)L1:(int)L2;'

#### **-Ont=9**

Disable optimization of 'i=i;'

#### **-Ont=8**

Disable optimization of empty switch statement

#### **-Ont=7**

Disable optimization of '(long) (char) L' into 'L'

#### **-Ont=1**

Disable optimization of 'a || 0' into 'a'

#### **-Ont=0**

Disable optimization of 'a && 1' into 'a'

#### **-Ont=/**

Disable optimization of 'a/1' into 'a'

#### **-Ont=-**

Disable optimization of 'a-0' into 'a'

#### **-Ont=+**

Disable optimization of 'a+0' into 'a'

#### **-Ont=\***

Disable optimization of 'a\*1' into 'a'

#### **-Ont=&**

Disable optimization of 'a&0' into '0'

**-Ont=%**

Disable optimization of 'a%1' into '0'

**Example**

```
fibonacci.c -Ont
```

## **-OnX: Disable Frame Pointer Optimization**

### **Group**

OPTIMIZATIONS

### **Scope**

Function

### **Syntax**

-OnX

### **Arguments**

None

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

Disable converting stack pointer relative accesses into X relative. The frame optimizer tries to convert all SP relative accesses (local variables, spills) into shorter and faster X relative accesses. In addition, the value of H:X is traced and useless TSX and AIX instructions are removed. Switch the frame optimizer off with -OnX to facilitate debugging.

### **Example**

-OnX

Disables frame pointer optimizations



## -Or: Allocate Local Variables into Registers

### Group

OPTIMIZATIONS

### Scope

Function

### Syntax

-Or

### Arguments

None

### Default

None

### Defines

\_\_OPTIMIZE\_REG\_\_

### Pragmas

None

### Description

Allocate local variables (`char` or `int`) in registers. The number of local variables allocated in registers depends on the number of available registers. This option is useful when using variables as loop counters or switch selectors or if the processor requires register operands for multiple operations (e.g., RISC processors). Compiling with this option may increase your code size (spill and merge code).

---

**NOTE** This optimization may result in code that could be very hard to debug at the High-level Language level.

---

---

**NOTE** This optimization will not take effect for some backends. For some backends the code does not change.

---

## Compiler Options

### Compiler Option Details

---

#### Listing 5.55 Example

---

```
-Or
int main(void) {
    int a,b;
    return a + b;
}
```

Case (1) without option -Or (pseudo code):

```
tmp1 LD    a
tmp2 LD    b
tmp3 ADD   tmp1,tmp2
      RET   tmp3
```

Case (2)with option -Or (pseudo code):

```
tmp1 ADD a,b
RET tmp1
```

---

## -Ou and -Onu: Optimize Dead Assignments

### Group

OPTIMIZATIONS

### Scope

Function

### Syntax

-O(u|nu)

### Arguments

None

### Default

Optimization enabled for functions containing no inline assembler code

### Defines

None

### Pragmas

None

### Description

Optimize dead assignments. Assignments to local variables, not referenced later, are removed by the Compiler.

There are three possible settings for this option: -Ou is given, -Onu is given, or neither of these options is given:

-Ou: Always optimize dead assignments (even if HLI is present in current function). Inline assembler accesses are not considered.

---

**NOTE** This option is not safe when accesses to local variables are contained in inline assembler code.

---

-Onu: The optimization does not take place. This generates the best possible debug information. The code is larger and slower than without -onu.

## Compiler Options

### Compiler Option Details

---

None of the options given: Optimize dead assignments if HLI is not present in the current function.

---

**NOTE** The compiler is not aware of `longjmp()` or `setjmp()` calls. These functions, those that are similar, may generate a control flow which is not recognized by the compiler. Therefore, be sure to either not use local variables in functions using `longjmp()` or `setjmp()` or switch this optimization off by using `-Onu`.

---

---

**NOTE** Dead assignments to `volatile` declared global objects are never optimized.

---

### Example

```
void main(int x) {
    f(x);
    x = 1; /* this assignment is dead and is
           removed if -Ou is active */
}
```

## **-Pe: Preprocessing Escape Sequences in Strings**

### **Group**

LANGUAGE

### **Scope**

Compilation Unit

### **Syntax**

-Pe

### **Arguments**

None

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

If escape sequences are used in macros, they are handled in an include directive similar to the way they are handled in a `printf()` instruction:

```
#define STRING "C:\myfile.h"  
#include STRING
```

produces an error:

```
>> Illegal escape sequence
```

and used in:

```
printf(STRING);
```

produces a carriage return with line feed:

```
C:
```

## Compiler Options

### Compiler Option Details

---

myfile

If the `-Pe` option is used, escape sequences are ignored in strings that contain a DOS drive letter ('a' – 'z', 'A' – 'Z') followed by a colon ':' and a backslash '\.

When the `-Pe` option is enabled, the Compiler handles strings in include directives differently from other strings. Escape sequences in include directive strings are not evaluated.

The following example:

```
#include "C:\names.h"
```

results in exactly the same include filename as in the source file ("C:\names.h"). If the filename appears in a macro, the Compiler does not distinguish between filename usage and normal string usage with escape sequence. This occurs because the macro `STRING` has to be the same for the include and the `printf()` call, as shown below:

```
#define STRING "C:\n.h"
#include STRING /* means: "C:\n.h" *

void main(void) {
    printf(STRING); /* means: "C:", new line and ".h" */
}
```

This option may be used to use macros for include files. This prevents escape sequence scanning in strings if the string starts with a DOS drive letter ('a' – 'z', 'A' – 'Z') followed by a colon ':' and a backslash '\ '. With the option set, the above example includes the 'C:\n.h' file and calls `printf()` with "C:\n.h").

### Example

`-Pe`

## **-Pio: Include Files Only Once**

### **Group**

INPUT

### **Scope**

Compilation Unit

### **Syntax**

-Pio

### **Arguments**

None

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

Includes every header file only once. Whenever the compiler reaches an `#include` directive, it checks if this file to be included was already read. If so, the compiler ignores the `#include` directive. It is common practice to protect header files from multiple inclusion by conditional compilation, as shown below:

---

```
/* Header file myfile.h */
#ifndef _MY_FILE_H_
#define _MY_FILE_H_

/* .... content .... */
#endif /* _MY_FILE_H_ */
```

---

When `#ifndef` and `#define` directives are issued, header file content is read only once even when the header file is included several times. This solves many

## Compiler Options

### *Compiler Option Details*

---

problems as C-language protocol does not allow you to define structures (such as enums or typedefs) more than once.

When all header files are protected this way, this option can safely accelerate the compilation.

This option must not be used when a header file must be included twice, e.g., the file contains macros which are set differently at the different inclusion times. In those instances, `#pragma ONCE: Include Once` is used to accelerate the inclusion of safe header files which do not contain macros of that nature.

### **Example**

```
-Pio
```

### **See also**

`#pragma ONCE: Include Once`



## **-Prod: Specify Project File at Startup**

### **Group**

Startup - This option cannot be specified interactively.

### **Scope**

None

### **Syntax**

`-Prod=<file>`

### **Arguments**

`<file>`: name of a project or project directory

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

This option can only be specified at the command line while starting the application. It cannot be specified in any other circumstances, including the `default.env` file, the command line or whatever.

When this option is given, the application opens the file as a configuration file.

When `<file>` names only a directory instead of a file, the default name `project.ini` is appended. When the loading fails, a message box appears.

### **Example**

```
compiler.exe -prod=project.ini
```

Use the compiler executable name instead of "compiler".

### **See also**

Local Configuration File (usually `project.ini`)

## **-Qvtp: Qualifier for Virtual Table Pointers**

### **Group**

CODE GENERATION

### **Scope**

Application

### **Syntax**

`-Qvtp (none | far | near | rom | uni | paged)`

### **Arguments**

None

### **Default**

`-Qvtpnone`

### **Defines**

None

### **Pragmas**

None

### **Description**

Using a virtual function in C++ requires an additional pointer to virtual function tables. This pointer is not accessible and is generated by the compiler in every class object when virtual function tables are associated.

---

**NOTE** It is useless to specify a qualifier which is not supported by the Backend (see Backend), e.g., using a `'far'` qualifier if the Backend or CPU does not support any `__far` data accesses.

---

### **Example**

`-QvtpFar`

This sets the qualifier for virtual table pointers to `__far` enabling the virtual tables to be placed into a `__FAR_SEG` segment (if the Backend or CPU supports `__FAR_SEG` segments).

## **-Rp (-Rpe, -Rpt): Large Return Value Type**

### **Group**

OPTIMIZATIONS

### **Scope**

Application

### **Syntax**

-Rp (t | e)

### **Arguments**

t: Pass the large return value by pointer

e: Pass the large return value with temporary elimination

### **Default**

-Rpe

### **Defines**

None

### **Pragmas**

None

### **Description**

This option is supported by the Compiler even though returning a ‘large’ return value may be not as efficient as using an additional pointer. The Compiler introduces an additional parameter for the return value if the return value could not be passed in registers.

Consider the following code in Listing 5.56

#### **Listing 5.56 Example code**

---

```
typedef struct { int i[10]; } S;  
  
S F(void);  
S s;
```

---

## Compiler Options

### Compiler Option Details

---

```
void main(void) {  
    s = F();  
}
```

---

In the above case, with `-Rpt`, the code will appear as in Listing 5.57:

#### Listing 5.57 Pass the large return value by pointer

---

```
void main(void) {  
    S tmp;  
  
    F(&tmp);  
    s = tmp; /* struct copy */  
}
```

---

The above approach is always correct but not efficient. The better solution is to pass the destination address directly to the callee making it unnecessary to declare a temporary and a struct copy in the caller (i.e., `-Rpe`), as shown below (Listing 5.58):

#### Listing 5.58 Pass the large return value with temporary elimination

---

```
void main(void) {  
    F(&s);  
}
```

---

The above example may produce incorrect results for rare cases, e.g., if the `F()` function returns something overlapping of `'s'`. Because it is not possible for the Compiler to detect such rare cases, two options are provided: the `-Rpt` (always correct, but inefficient), or `-Rpe` (efficient) options.

## -T: Flexible Type Management

### Group

LANGUAGE.

### Scope

Application

### Syntax

-T<Type Format>

### Arguments

<Type Format>: See below

### Default

Depends on target, see Compiler Backend

### Defines

To deal with different type sizes, one in the following define groups in Listing 5.59 will be predefined by the Compiler:

#### Listing 5.59 define groups

---

```
__CHAR_IS_SIGNED__
__CHAR_IS_UNSIGNED__

__CHAR_IS_8BIT__
__CHAR_IS_16BIT__
__CHAR_IS_32BIT__
__CHAR_IS_64BIT__

__SHORT_IS_8BIT__
__SHORT_IS_16BIT__
__SHORT_IS_32BIT__
__SHORT_IS_64BIT__

__INT_IS_8BIT__
__INT_IS_16BIT__
__INT_IS_32BIT__
```

## Compiler Options

### Compiler Option Details

---

\_\_INT\_IS\_64BIT\_\_  
\_\_ENUM\_IS\_8BIT\_\_  
\_\_ENUM\_IS\_16BIT\_\_  
\_\_ENUM\_IS\_32BIT\_\_  
\_\_ENUM\_IS\_64BIT\_\_  
\_\_ENUM\_IS\_SIGNED\_\_  
\_\_ENUM\_IS\_UNSIGNED\_\_  
\_\_PLAIN\_BITFIELD\_IS\_SIGNED\_\_  
\_\_PLAIN\_BITFIELD\_IS\_UNSIGNED\_\_  
\_\_LONG\_IS\_8BIT\_\_  
\_\_LONG\_IS\_16BIT\_\_  
\_\_LONG\_IS\_32BIT\_\_  
\_\_LONG\_IS\_64BIT\_\_  
\_\_LONG\_LONG\_IS\_8BIT\_\_  
\_\_LONG\_LONG\_IS\_16BIT\_\_  
\_\_LONG\_LONG\_IS\_32BIT\_\_  
\_\_LONG\_LONG\_IS\_64BIT\_\_  
\_\_FLOAT\_IS\_IEEE32\_\_  
\_\_FLOAT\_IS\_IEEE64\_\_  
\_\_FLOAT\_IS\_DSP\_\_  
\_\_DOUBLE\_IS\_IEEE32\_\_  
\_\_DOUBLE\_IS\_IEEE64\_\_  
\_\_DOUBLE\_IS\_DSP\_\_  
\_\_LONG\_DOUBLE\_IS\_IEEE32\_\_  
\_\_LONG\_DOUBLE\_IS\_IEEE64\_\_  
\_\_LONG\_DOUBLE\_IS\_DSP\_\_  
\_\_LONG\_LONG\_DOUBLE\_IS\_IEEE32\_\_  
\_\_LONG\_LONG\_DOUBLE\_IS\_IEEE64\_\_  
\_\_LONG\_LONG\_DOUBLE\_DSP\_\_  
\_\_VTAB\_DELTA\_IS\_8BIT\_\_  
\_\_VTAB\_DELTA\_IS\_16BIT\_\_  
\_\_VTAB\_DELTA\_IS\_32BIT\_\_  
\_\_VTAB\_DELTA\_IS\_64BIT\_\_  
\_\_PTRMBR\_OFFSET\_IS\_8BIT\_\_  
\_\_PTRMBR\_OFFSET\_IS\_16BIT\_\_  
\_\_PTRMBR\_OFFSET\_IS\_32BIT\_\_

\_\_PTRMBR\_OFFSET\_IS\_64BIT\_\_

### Pragmas

None

### Description

This option allows configurable type settings. The syntax of the option is:  
 -T{<type><format>}

For <type>, one of the keys listed in Table 5.9 may be specified:

**Table 5.9 Data Type Keys**

Type	Key
char	'c'
short	's'
int	'i'
long	'L'
long long	'LL'
float	'f'
double	'd'
long double	'Ld'
long long double	'LLd'
enum	'e'
sign plain bitfield	'b'
virtual table delta size	'vtd'
pointer to member offset size	'pmo'

---

**NOTE** Keys are not case-sensitive, e.g., both 'f' and 'F' may be used for the type "float".

---

## Compiler Options

### Compiler Option Details

---

The sign of the type 'char,' or of the enumeration type, may be changed with a prefix placed before the key for the char key. See Table 5.10.

**Table 5.10 Keys for Signed and Unsigned Prefixes**

Sign prefix	Key
signed	's'
unsigned	'u'

The sign of the type 'plain bitfield type' is changed with the options shown in Table 5.11. Plain bitfields are bitfields defined or declared without an explicit signed or unsigned qualifier, e.g., 'int field:3'. Using this option, you can specify if the 'int' in the previous example is handled as 'signed int' or as 'unsigned int'. Note that this option may not be available on all targets. Also the default setting may vary. Refer to Sign of Plain Bitfields.

**Table 5.11 Keys for Signed and Unsigned Bitfield Prefixes**

Sign prefix	Key
plain signed bitfield	'bs'
plain unsigned bitfield	'bu'

For <format>, one of the keys in Table 5.12 can be specified.

**Table 5.12 Data Format Specifier Keys**

Format	Key
8-bit integral	'1'
16-bit integral	'2'
24-bit integral	'3'
32-bit integral	'4'
64-bit integral	'8'
IEEE32 floating	'2'
IEEE64 floating	'4'
DSP (32-bit)	'0'



Not all formats may be available for a target. See Backend for supported formats.

---

**NOTE** At least one type for each basic size (1, 2, 4 bytes) has to be available. It is not allowed that no type of any sort is not set to at least a size of one. See Backend for default settings.

---

---

**NOTE** Enumeration types have the type 'signed int' by default for ANSI-C compliance.

---

The `-Tpmo` option allows you to change the pointer to a member offset value type. The default setting is 16 bits. The pointer to the member offset is used for C++ pointer to members only.

`-Tsc` sets 'char' to 'signed char'  
and `-Tuc` sets 'char' to 'unsigned char'

### Listing 5.60 Example

---

`-Tsc1s2i2L4LL4f2d4Ld4LLd4e2` denotes:  
signed char with 8 bits (`s1`)  
short and int with 16 bits (`s2i2`)  
long, long long with 32 bits (`L4LL4`)  
float with IEEE32 (`f2`)  
double, long double and long long double with IEEE64 (`d4Ld4LL4`)  
enum with 16 bits (signed) (`e2`)

---

### Listing 5.61 Restrictions

---

For integrity and compliance to ANSI, the following two rules must be true:

---

<code>sizeof(char)</code>	<code>&lt;= sizeof(short)</code>
<code>sizeof(short)</code>	<code>&lt;= sizeof(int)</code>
<code>sizeof(int)</code>	<code>&lt;= sizeof(long)</code>
<code>sizeof(long)</code>	<code>&lt;= sizeof(long long)</code>
<code>sizeof(float)</code>	<code>&lt;= sizeof(double)</code>
<code>sizeof(double)</code>	<code>&lt;= sizeof(long double)</code>
<code>sizeof(long double)</code>	<code>&lt;= sizeof(long long double)</code>

---

---

**NOTE** It is not allowed to set `char` to 16 bits and `int` to 8 bits.

---

## Compiler Options

### Compiler Option Details

---

Be careful if you change type sizes. Type sizes must be consistent over the whole application. The libraries delivered with the Compiler are compiled with the standard type settings.

Also be careful if you change the type sizes for under or overflows, e.g., assigning a value too large to an object which is smaller now, as shown in the following example:

```
int i; /* -Ti1 int has been set to 8 bits! */
i = 0x1234; /* i will set to 0x34! */
```

#### Examples:

Setting the size of char to 16 bits:

```
-Tc2
```

Setting the size of char to 16 bits and plain char is signed:

```
-Tsc2
```

Setting char to 8 bits and unsigned, int to 32 bits and long long to 32 bits:

```
-Tuc1i4LL4
```

Setting float to IEEE32 and double to IEEE64:

```
-Tf2d4
```

The `-Tvtδ` option allows you to change the delta value type inside virtual function tables. The default setting is 16-bit.

Another way to set this option is using the dialog box in the Graphical User Interface:

#### See also

Sign of Plain Bitfields

## **-V: Prints the Compiler Version**

### **Group**

VARIOUS

### **Scope**

None

### **Syntax**

-V

### **Arguments**

None

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

Prints the internal subversion numbers of the component parts of the Compiler and the location of current directory.

---

**NOTE** This option can determine the current directory of the Compiler.

---

### **Example**

-V produces the following list:

```
Directory: \software\sources\c
ANSI-C Front End, V5.0.1, Date Jan 01 2005
Tree CSE Optimizer, V5.0.1, Date Jan 01 2005
Back End V5.0.1, Date Jan 01 2005
```

## **-View: Application Standard Occurrence**

### **Group**

HOST

### **Scope**

Compilation Unit

### **Syntax**

`-View<kind>`

### **Arguments**

`<kind>` is one of:

Window: Application window has default window size

Min: Application window is minimized

Max: Application window is maximized

Hidden: Application window is not visible (only if arguments)

### **Default**

Application started with arguments: Minimized

Application started without arguments: Window

### **Defines**

None

### **Pragmas**

None

### **Description**

The application (e.g., Linker, Compiler, ...) is started as a normal window if no arguments are given. If the application is started with arguments (e.g., from the maker to compile or link a file), then the application runs minimized to allow batch processing.

You can specify the behavior of the application using this option. Using `-ViewWindow`, the application is visible with its normal window. Using `-ViewMin` the application is visible iconified (in the task bar). Using `-ViewMax` the application is visible maximized (filling the whole screen). Using `-ViewHidden` the application processes arguments (e.g., files to be compiled or

linked) completely invisible in the background (no window or icon visible in the task bar). However, if you are using the -N: Display Notify Box option, a dialog box is still possible.

**Example**

```
C:\Freescale\linker.exe -ViewHidden fibo.prm
```

## **-WErrFile: Create "err.log" Error File**

### **Group**

MESSAGES

### **Scope**

Compilation Unit

### **Syntax**

`-WErrFile(On|Off)`

### **Arguments**

None

### **Default**

`err.log` is created or deleted

### **Defines**

None

### **Pragmas**

None

### **Description**

The error feedback to the tools that are called is done with a return code. In 16-bit window environments, this was not possible. In the error case, an "err.log" file, with the numbers of errors written into it, was used to signal an error. To state no error, the "err.log" file was deleted. Using UNIX or WIN32, there is now a return code available. The "err.log" file is no longer needed when only UNIX or WIN32 applications are involved.

NOTE: The error file must be created in order to signal any errors if you use a 16-bit maker with this tool.

### **Example**

```
-WErrFileOn
```

The `err.log` file is created or deleted when the application is finished.

```
-WErrFileOff
```

The existing `err.log` file is not modified.

**See also**

- WStdout: Write to Standard Output
- WOutFile: Create Error Listing File

## **-Wmsg8x3: Cut filenames in Microsoft Format to 8.3**

### **Group**

MESSAGES

### **Scope**

Compilation Unit

### **Syntax**

-Wmsg8x3

### **Arguments**

None

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

Some editors (early versions of WinEdit) expect the filename in the Microsoft message format (8.3 format). That means the filename can have up to eight characters and no more than a three-character extension. Longer filenames are possible when you use Win95 or WinNT. This option truncates the filename to the 8.3 format.

### **Example**

```
x:\mysourcefile.c(3): INFORMATION C2901: Unrolling loop
```

With the option -Wmsg8x3 set, the above message is:

```
x:\mysource.c(3): INFORMATION C2901: Unrolling loop
```

### **See also**

-WmsgFi (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode

-WmsgFb (-WmsgFbi, -WmsgFbm): Set Message File Format for Batch Mode



## **-WmsgCE: RGB Color for Error Messages**

### **Group**

MESSAGES

### **Scope**

Function

### **Syntax**

`-WmsgCE<RGB>`

### **Arguments**

`<RGB>`: 24-bit RGB (red green blue) value

### **Default**

`-WmsgCE16711680 (rFF g00 b00, red)`

### **Defines**

None

### **Pragmas**

None

### **Description**

This option changes the error message color. The specified value must be an RGB (Red-Green-Blue) value and must also be specified in decimal.

### **Example**

`-WmsgCE255` changes the error messages to blue

## **-WmsgCF: RGB Color for Fatal Messages**

### **Group**

MESSAGES

### **Scope**

Function

### **Syntax**

`-WmsgCF<RGB>`

### **Arguments**

`<RGB>`: 24-bit RGB (red green blue) value

### **Default**

`-WmsgCF8388608 (r80 g00 b00, dark red)`

### **Defines**

None

### **Pragmas**

None

### **Description**

This option changes the color of a fatal message. The specified value must be an RGB (Red-Green-Blue) value and must also be specified in decimal.

### **Example**

`-WmsgCF255` changes the fatal messages to blue

## **-WmsgCI: RGB Color for Information Messages**

### **Group**

MESSAGES

### **Scope**

Function

### **Syntax**

`-WmsgCI<RGB>`

### **Arguments**

`<RGB>`: 24-bit RGB (red green blue) value

### **Default**

`-WmsgCI32768 (r00 g80 b00, green)`

### **Defines**

None

### **Pragmas**

None

### **Description**

This option changes the color of an information message. The specified value must be an RGB (Red-Green-Blue) value and must also be specified in decimal.

### **Example**

`-WmsgCI255` changes the information messages to blue

## **-WmsgCU: RGB Color for User Messages**

### **Group**

MESSAGES

### **Scope**

Function

### **Syntax**

`-WmsgCU<RGB>`

### **Arguments**

`<RGB>`: 24-bit RGB (red green blue) value

### **Default**

`-WmsgCU0 (r00 g00 b00, black)`

### **Defines**

None

### **Pragmas**

None

### **Description**

This option changes the color of a user message. The specified value must be an RGB (Red-Green-Blue) value and must also be specified in decimal.

### **Example**

`-WmsgCU255` changes the user messages to blue

## **-WmsgCW: RGB Color for Warning Messages**

### **Group**

MESSAGES

### **Scope**

Function

### **Syntax**

`-WmsgCW<RGB>`

### **Arguments**

`<RGB>`: 24-bit RGB (red green blue) value

### **Default**

`-WmsgCW255 (r00 g00 bFF, blue)`

### **Defines**

None

### **Pragmas**

None

### **Description**

This option changes the color of a warning message. The specified value must be an RGB (Red-Green-Blue) value and must also be specified in decimal.

### **Example**

`-WmsgCW0` changes the warning messages to black

## **-WmsgFb (-WmsgFbi, -WmsgFbm): Set Message File Format for Batch Mode**

### **Group**

MESSAGES

### **Scope**

Compilation Unit

### **Syntax**

`-WmsgFb [v|m]`

### **Arguments**

v: Verbose format  
m: Microsoft format

### **Default**

`-WmsgFbm`

### **Defines**

None

### **Pragmas**

None

### **Description**

You can start the Compiler with additional arguments (e.g., files to be compiled together with Compiler options). If the Compiler has been started with arguments (e.g., from the Make Tool or with the `'%f'` argument from the CodeWright IDE), the Compiler compiles the files in a batch mode. No Compiler window is visible and the Compiler terminates after job completion.

If the Compiler is in batch mode, the Compiler messages are written to a file instead of to the screen. This file contains only the compiler messages (see examples below).

The Compiler uses a Microsoft message format to write the Compiler messages (errors, warnings, information messages) if the compiler is in batch mode.

This option changes the default format from the Microsoft format (only line information) to a more verbose error format with line, column, and source information.

---

**NOTE** Using the verbose message format may slow down the compilation because the compiler has to write more information into the message file.

---

### Listing 5.62 Example

---

```
void foo(void) {
    int i, j;
    for(i=0;i<1;i++);
}
```

The Compiler may produce the following file if it is running in batch mode (e.g., started from the Make tool):

```
X:\C.C(3): INFORMATION C2901: Unrolling loop
X:\C.C(2): INFORMATION C5702: j: declared in function foo but not
referenced
```

Setting the format to verbose, more information is stored in the file:

```
-WmsgFbv
>> in "X:\C.C", line 3, col 2, pos 33
    int i, j;

    for(i=0;i<1;i++);

    ^
INFORMATION C2901: Unrolling loop
>> in "X:\C.C", line 2, col 10, pos 28
void foo(void) {

    int i, j;

    ^
INFORMATION C5702: j: declared in function foo but not referenced
```

---

### See also

ERRORFILE: Error filename Specification environment variable  
-WmsgFi (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode

## **-WmsgFi (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode**

### **Group**

MESSAGES

### **Scope**

Compilation Unit

### **Syntax**

`-WmsgFi [v|m]`

### **Arguments**

v: Verbose format  
m: Microsoft format

### **Default**

`-WmsgFiv`

### **Defines**

None

### **Pragmas**

None

### **Description**

The Compiler operates in the interactive mode (that is, a window is visible) if it is started without additional arguments (e.g., files to be compiled together with Compiler options).

The Compiler uses the verbose error file format to write the Compiler messages (errors, warnings, information messages).

This option changes the default format from the verbose format (with source, line and column information) to the Microsoft format (only line information).



---

**NOTE** Using the Microsoft format may speed up the compilation because the compiler has to write less information to the screen.

---

### **Example**

---

```
void foo(void) {  
    int i, j;  
    for(i=0;i<1;i++);  
}
```

The Compiler may produce the following error output in the Compiler window if it is running in interactive mode:

Top: X:\C.C

Object File: X:\C.O

```
>> in "X:\C.C", line 3, col 2, pos 33
```

```
    int i, j;
```

```
        for(i=0;i<1;i++);
```

```
        ^
```

INFORMATION C2901: Unrolling loop

Setting the format to Microsoft, less information is displayed:

-WmsgFim

Top: X:\C.C

Object File: X:\C.O

X:\C.C(3): INFORMATION C2901: Unrolling loop

---

### **See also**

ERRORFILE: Error filename Specification

-WmsgFb (-WmsgFbi, -WmsgFbm): Set Message File Format for Batch Mode

## **-WmsgFob: Message Format for Batch Mode**

### **Group**

MESSAGES

### **Scope**

Function

### **Syntax**

`-WmsgFob<string>`

### **Arguments**

`<string>`: format string (see below).

### **Default**

`-WmsgFob"% "%f%e%" (%l) : %K %d: %m\n"`

### **Defines**

None

### **Pragmas**

None

### **Description**

This option modifies the default message format in batch mode. The formats listed in Table 5.13 are supported (assuming that the source file is `X:\Freescale\mysourcefile.cpph`):

**Table 5.13 Message Format Specifiers**

<b>Format</b>	<b>Description</b>	<b>Example</b>
<code>%s</code>	Source Extract	
<code>%p</code>	Path	<code>X:\Freescale\</code>
<code>%f</code>	Path and name	<code>X:\Freescale\mysourcefile</code>
<code>%n</code>	filename	<code>mysourcefile</code>

**Table 5.13 Message Format Specifiers (continued)**

Format	Description	Example
%e	Extension	.cpph
%N	File (8 chars)	mysource
%E	Extension (3 chars)	.cpp
%l	Line	3
%c	Column	47
%o	Pos	1234
%K	Uppercase kind	ERROR
%k	Lowercase kind	error
%d	Number	C1815
%m	Message	text
%%	Percent	%
\n	New line	
%"	A " if the filename, if the path or the extension contains a space	
%'	A ' if the filename, the path or the extension contains a space	

**Example**

```
-WmsgFob"%f%e(%l): %k %d: %m\n"
```

Produces a message in the following format:

```
X:\C.C(3): information C2901: Unrolling loop
```

**See also**

ERRORFILE: Error filename Specification

-WmsgFb (-WmsgFbi, -WmsgFbm): Set Message File Format for Batch Mode

-WmsgFi (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode

-WmsgFonp: Message Format for no Position Information

-WmsgFoi: Message Format for Interactive Mode

## **-WmsgFoi: Message Format for Interactive Mode**

### **Group**

MESSAGES

### **Scope**

Function

### **Syntax**

`-WmsgFoi<string>`

### **Arguments**

`<string>`: format string (See below.)

### **Default**

```
-WmsgFoi"\n>> in \"%f%e\", line %l, col >>%c, pos  
%o\n%s\n%K %d: %m\n"
```

### **Defines**

None

### **Pragmas**

None

### **Description**

This option modifies the default message format in interactive mode. The formats listed in Table 5.14 are supported (assuming that the source file is `X:\Freescale\mysourcefile.cpph`):

**Table 5.14 Message Format Specifiers**

<b>Format</b>	<b>Description</b>	<b>Example</b>
<code>%s</code>	Source Extract	
<code>%p</code>	Path	<code>X:\sources\</code>
<code>%f</code>	Path and name	<code>X:\sources\mysourcefile</code>
<code>%n</code>	filename	<code>mysourcefile</code>

**Table 5.14 Message Format Specifiers (continued)**

Format	Description	Example
%e	Extension	.cpph
%N	File (8 chars)	mysource
%E	Extension (3 chars)	.cpp
%l	Line	3
%c	Column	47
%o	Pos	1234
%K	Uppercase kind	ERROR
%k	Lowercase kind	error
%d	Number	C1815
%m	Message	text
%%	Percent	%
\n	New line	
%"	A " if the filename, if the path or the extension contains a space.	
%'	A ' if the filename, the path or the extension contains a space	

**Example**

```
-WmsgFoi"%f%e(%l): %k %d: %m\n"
```

Produces a message in following format

```
X:\C.C(3): information C2901: Unrolling loop
```

**See also**

ERRORFILE: Error filename Specification

-WmsgFb (-WmsgFbi, -WmsgFbm): Set Message File Format for Batch Mode

-WmsgFi (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode

-WmsgFonp: Message Format for no Position Information

-WmsgFob: Message Format for Batch Mode

## **-WmsgFonf: Message Format for no File Information**

### **Group**

MESSAGES

### **Scope**

Function

### **Syntax**

`-WmsgFonf<string>`

### **Arguments**

`<string>`: format string (See below.)

### **Default**

`-WmsgFonf "%K %d: %m\n"`

### **Defines**

None

### **Pragmas**

None

### **Description**

Sometimes there is no file information available for a message (e.g., if a message is not related to a specific file). Then the message format string defined by `<string>` is used. Table 5.15 lists the supported formats.

**Table 5.15 Message Format Specifiers**

<b>Format</b>	<b>Description</b>	<b>Example</b>
<code>%K</code>	Uppercase kind	ERROR
<code>%k</code>	Lowercase kind	error
<code>%d</code>	Number	C1815

**Table 5.15 Message Format Specifiers (*continued*)**

<b>Format</b>	<b>Description</b>	<b>Example</b>
%m	Message	text
%%	Percent	%
\n	New line	
%"	A " if the filename, if the path or the extension contains a space	
%'	A ' if the filename, the path or the extension contains a space	

**Example**

```
-WmsgFonf "%k %d: %m\n"
```

Produces a message in the following format:

```
information L10324: Linking successful
```

**See also**

- ERRORFILE: Error filename Specification
- WmsgFb (-WmsgFbi, -WmsgFbm): Set Message File Format for Batch Mode
- WmsgFi (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode
- WmsgFonp: Message Format for no Position Information
- WmsgFoi: Message Format for Interactive Mode

## **-WmsgFonp: Message Format for no Position Information**

### **Group**

MESSAGES

### **Scope**

Function

### **Syntax**

`-WmsgFonp<string>`

### **Arguments**

`<string>`: format string (See below.)

### **Default**

`-WmsgFonp "%f%e%": %K %d: %m\n"`

### **Defines**

None

### **Pragmas**

None

### **Description**

Sometimes there is no position information available for a message (e.g., if a message not related to a certain position). Then the message format string defined by `<string>` is used. Table 5.16 lists the supported formats.

**Table 5.16 Message Format specifiers**

<b>Format</b>	<b>Description</b>	<b>Example</b>
<code>%K</code>	Uppercase type	ERROR
<code>%k</code>	Lowercase type	error
<code>%d</code>	Number	C1815
<code>%m</code>	Message	text



Table 5.16 Message Format specifiers (*continued*)

Format	Description	Example
%%	Percent	%
\n	New line	
%"	A " if the filename, if the path or the extension contains a space	
%'	A ' if the filename, the path, or the extension contains a space	

### Example

```
-WmsgFonf "%k %d: %m\n"
```

Produces a message in following format:

```
information L10324: Linking successful
```

### See also

ERRORFILE: Error filename Specification

-WmsgFb (-WmsgFbi, -WmsgFbm): Set Message File Format for Batch Mode

-WmsgFi (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode

-WmsgFonp: Message Format for no Position Information

-WmsgFoi: Message Format for Interactive Mode

## **-WmsgNe: Number of Error Messages**

### **Group**

MESSAGES

### **Scope**

Compilation Unit

### **Syntax**

`-WmsgNe<number>`

### **Arguments**

`<number>`: Maximum number of error messages

### **Default**

50

### **Defines**

None

### **Pragmas**

None

### **Description**

This option sets the number of error messages that are to be displayed while the Compiler is processing.

---

**NOTE** Subsequent error messages which depend upon a previous error message may not process correctly.

---

### **Example**

`-WmsgNe2`

Stops compilation after two error messages

### **See also**

-WmsgNi: Number of Information Messages

-WmsgNw: Number of Warning Messages

## **-WmsgNi: Number of Information Messages**

### **Group**

MESSAGES

### **Scope**

Compilation Unit

### **Syntax**

`-WmsgNi <number>`

### **Arguments**

`<number>`: Maximum number of information messages

### **Default**

50

### **Defines**

None

### **Pragmas**

None

### **Description**

This option sets the amount of information messages that are logged.

### **Example**

```
-WmsgNi 10
```

Ten information messages logged

### **See also**

-WmsgNe: Number of Error Messages

-WmsgNw: Number of Warning Messages

## **-WmsgNu: Disable User Messages**

### **Group**

MESSAGES

### **Scope**

None

### **Syntax**

`-WmsgNu [= { a | b | c | d | e } ]`

### **Arguments**

- a: Disable messages about include files
- b: Disable messages about reading files
- c: Disable messages about generated files
- d: Disable messages about processing statistics
- e: Disable informal messages

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

The application produces messages that are not in the following normal message categories: WARNING, INFORMATION, ERROR, or FATAL. This option disables messages that are not in the normal message category by reducing the amount of messages, and simplifying the error parsing of other tools.

- a: Disables the application from generating information about all included files.
- b: Disables messages about reading files (e.g., the files used as input) are disabled.
- c: Disables messages informing about generated files.
- d: Disables information about statistics (e.g., code size, RAM or ROM usage and

so on).  
e: Disables informal messages (e.g., memory model, floating point format, ...).

---

**NOTE** Depending on the application, the Compiler may not recognize all suboptions.  
In this case they are ignored for compatibility.

---

**Example**

`-WmsgNu=c`

## **-WmsgNw: Number of Warning Messages**

### **Group**

MESSAGES

### **Scope**

Compilation Unit

### **Syntax**

`-WmsgNw<number>`

### **Arguments**

`<number>`: Maximum number of warning messages

### **Default**

50

### **Defines**

None

### **Pragmas**

None

### **Description**

This option sets the number of warning messages.

### **Example**

`-WmsgNw15`

Fifteen warning messages logged

### **See also**

-WmsgNe: Number of Error Messages

-WmsgNi: Number of Information Messages

## **-WmsgSd: Setting a Message to Disable**

### **Group**

MESSAGES

### **Scope**

Function

### **Syntax**

`-WmsgSd<number>`

### **Arguments**

`<number>`: Message number to be disabled, e.g., 1801

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

This option disables message from appearing in the error output.  
This option cannot be used in `#pragma OPTION: Additional Options`. Use this option only with `#pragma MESSAGE: Message Setting`.

### **Example**

```
-WmsgSd1801
```

Disables message for implicit parameter declaration

### **See also**

- WmsgSe: Setting a Message to Error
- WmsgSi: Setting a Message to Information
- WmsgSw: Setting a Message to Warning

## **-WmsgSe: Setting a Message to Error**

### **Group**

MESSAGES

### **Scope**

Function

### **Syntax**

`-WmsgSe<number>`

### **Arguments**

`<number>`: Message number to be an error, e.g., 1853

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

This option changes a message to an error message.

This option cannot be used in `#pragma OPTION: Additional Options`. Use this option only with `#pragma MESSAGE: Message Setting`.

### **Listing 5.63 Example**

---

```
COMPOTIONS=-WmsgSe1853
```

---

### **See also**

- WmsgSd: Setting a Message to Disable
- WmsgSi: Setting a Message to Information
- WmsgSw: Setting a Message to Warning



## **-WmsgSi: Setting a Message to Information**

### **Group**

MESSAGES

### **Scope**

Function

### **Syntax**

`-WmsgSi<number>`

### **Arguments**

`<number>`: Message number to be an information, e.g., 1853

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

This option sets a message to an information message.

This option cannot be used with `#pragma OPTION: Additional Options`. Use this option only with `#pragma MESSAGE: Message Setting`.

### **Example**

```
-WmsgSi1853
```

### **See also**

- WmsgSd: Setting a Message to Disable
- WmsgSe: Setting a Message to Error
- WmsgSw: Setting a Message to Warning

## **-WmsgSw: Setting a Message to Warning**

### **Group**

MESSAGES

### **Scope**

Function

### **Syntax**

`-WmsgSw<number>`

### **Arguments**

`<number>`: Error number to be a warning, e.g., 2901

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

This option sets a message to a warning message.

This option cannot be used with `#pragma OPTION: Additional Options`. Use this option only with `#pragma MESSAGE: Message Setting`.

### **Example**

```
-WmsgSw2901
```

### **See also**

- WmsgSd: Setting a Message to Disable
- WmsgSe: Setting a Message to Error
- WmsgSi: Setting a Message to Information

## **-WOutFile: Create Error Listing File**

### **Group**

MESSAGES

### **Scope**

Compilation Unit

### **Syntax**

`-WOutFile(On|Off)`

### **Arguments**

None

### **Default**

Error listing file is created

### **Defines**

None

### **Pragmas**

None

### **Description**

This option controls whether an error listing file should be created. The error listing file contains a list of all messages and errors that are created during processing. It is possible to obtain this feedback without an explicit file because the text error feedback can now also be handled with pipes to the calling application. The name of the listing file is controlled by the `ERRORFILE: Error filename Specification` environment variable.

### **Example**

`-WOutFileOn`

Error file is created as specified with `ERRORFILE`

`-WOutFileOff`

No error file created

## Compiler Options

*Compiler Option Details*

---

### See also

- WErrFile: Create "err.log" Error File
- WStdout: Write to Standard Output

## **-Wpd: Error for Implicit Parameter Declaration**

### **Group**

MESSAGES

### **Scope**

Function

### **Syntax**

-Wpd

### **Arguments**

None

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

This option prompts the Compiler to issue an ERROR message instead of a WARNING message when an implicit declaration is encountered. This occurs if the Compiler does not have a prototype for the called function.

This option helps to prevent parameter-passing errors, which can only be detected at runtime. It requires that each function that is called is prototyped before use. The correct ANSI behavior is to assume that parameters are correct for the stated call.

This option is the same as using `-WmsgSe1801`.

### **Example**

---

```
-Wpd  
  
main() {  
    char a, b;
```

## Compiler Options

### Compiler Option Details

---

```
    func(a, b); // <- Error here
}
func(a, b, c)
    char a, b, c;
{
    ...
}
```

---

### See also

Message C1801  
-WmsgSe: Setting a Message to Error

## **-WStdout: Write to Standard Output**

### **Group**

MESSAGES

### **Scope**

Compilation Unit

### **Syntax**

-WStdout (On|Off)

### **Arguments**

None

### **Default**

Output is written to `stdout`

### **Defines**

None

### **Pragmas**

None

### **Description**

The usual standard streams are available with Windows applications. Text written into them does not appear anywhere unless explicitly requested by the calling application. This option determines if error file text to the error file is also written into the `stdout` file.

### **Example**

-WStdoutOn

All messages written to `stdout`

-WErrFileOff

Nothing written to `stdout`

## Compiler Options

*Compiler Option Details*

---

### See also

- WErrFile: Create "err.log" Error File
- WOutFile: Create Error Listing File



## **-W1: No Information Messages**

### **Group**

MESSAGES

### **Scope**

Function

### **Syntax**

-W1

### **Arguments**

None

### **Default**

None

### **Defines**

None

### **Pragmas**

None

### **Description**

Inhibits printing INFORMATION messages. Only WARNINGS and ERROR messages are generated.

### **Example**

-W1

### **See also**

-WmsgNi: Number of Information Messages

## **-W2: No Information and Warning Messages**

**Group**

MESSAGES

**Scope**

Function

**Syntax**

-W2

**Arguments**

None

**Default**

None

**Defines**

None

**Pragmas**

None

**Description**

Suppresses all messages of type INFORMATION and WARNING. Only ERRORS are generated.

**Example**

-W2

**See also**

-WmsgNi: Number of Information Messages

-WmsgNw: Number of Warning Messages

# Compiler Predefined Macros

The ANSI standard for the C language requires the Compiler to predefine a couple of macros. The Compiler provides the predefined macros listed in Table 6.1.

**Table 6.1** Macros defined by the Compiler

Macro	Description
<code>__LINE__</code>	Line number in the current source file
<code>__FILE__</code>	Name of the source file where it appears
<code>__DATE__</code>	The date of compilation as a string
<code>__TIME__</code>	The time of compilation as a string
<code>__STDC__</code>	Set to 1 if the <code>-Ansi: Strict ANSI compiler option</code> has been given. Otherwise, additional keywords are accepted (not in the ANSI standard).

The following tables lists all Compiler defines with their associated names and options.

**NOTE** If these macros do not have a value, the Compiler treats them as if they had been defined as shown: `#define __HIWARE__`

It is also possible to log all Compiler predefined defines to a file using the `-Ldf: Log Predefined Defines to File compiler option`.

## Compiler Vendor Defines

Table 6.2 shows the defines identifying the Compiler vendor. Compilers in the USA may also be sold by ARCHIMEDES.

## Compiler Predefined Macros

### Product Defines

---

**Table 6.2 Compiler Vendor Identification Defines**

Name	Defined
<code>__HIWARE__</code>	always
<code>__MWERKS__</code>	always, set to 1

## Product Defines

Table 6.3 shows the Defines identifying the Compiler. The Compiler is a HI-CROSS+ Compiler (V5.0.x).

**Table 6.3 Compiler Identification Defines**

Name	Defined
<code>__PRODUCT_HICROSS_PLUS__</code>	defined for V5.0 Compilers
<code>__DEMO_MODE__</code>	defined if the Compiler is running in demo mode
<code>__VERSION__</code>	defined and contains the version number, e.g., it is set to 5013 for a Compiler V5.0.13, or set to 3140 for a Compiler V3.1.40

## Data Allocation Defines

The Compiler provides two macros that define how data is organized in memory: Little Endian (least significant byte first in memory) or Big Endian (most significant byte first in memory). The ‘Intel World’ uses Little Endian and the ‘Non-Intel World’ uses Big Endian.

The Compiler provides the “endian” macros listed in Table 6.4.

**Table 6.4 Compiler macros for defining “endianness”**

Name	Defined
<code>__LITTLE_ENDIAN__</code>	defined if the Compiler allocates in Little Endian order
<code>__BIG_ENDIAN__</code>	defined if the Compiler allocates in Big Endian order

The following example illustrates the difference between little and big endian (Listing 6.1).

**Listing 6.1 Little vs. big edian**

---

```
unsigned long L = 0x87654321;
unsigned short s = *(unsiged short*)&L; // BE: 0x8765,LE: 0x4321
unsigned char c = *(unsinged char*)&L; // BE: 0x87, LE: 0x21
```

---

## Various Defines for compiler option settings

The following table lists defines for miscellaneous compiler option settings.

**Table 6.5 Defines for Miscellaneous Compiler Option Settings**

Name	Defined
__STDC__	-Ansi
__TRIGRAPHS__	-Ci
__CNI__	-Cni
__OPTIMIZE_FOR_TIME__	-Ot
__OPTIMIZE_FOR_SIZE__	-Os

## Option Checking in C Code

You can also check the source to determine if an option is active. The EBNF syntax is:

```
OptionActive = __OPTION_ACTIVE__ ("string")
```

The above is used in the preprocessor and in C code, as shown:

**Listing 6.2 Using \_\_OPTION\_\_ to check for active options**

---

```
#if __OPTION_ACTIVE__("-W2")
    // option -W2 is set
#endif

void main(void) {
    int i;
    if (__OPTION_ACTIVE__("-or")) {
```

## Compiler Predefined Macros

### ANSI-C Standard Types 'size\_t', 'wchar\_t' and 'ptrdiff\_t' Defines

---

```
    i=2;
}
}
```

---

You can check all valid preprocessor options (e.g., options given at the command line, via the `default.env` or `project.ini` files, but not options added with the `#pragma OPTION: Additional Options`). You perform the same check in C code using `-Odocf` and `#pragma OPTION`.

As a parameter, only the option itself is tested and not a specific argument of an option.

See Listing 6.3 for a valid and an invalid use of `__OPTION_ACTIVE__`.

#### Listing 6.3 Using `__OPTION_ACTIVE__`

---

```
#if __OPTION_ACTIVE__("-D") /* true if any -d option is given */
#if __OPTION_ACTIVE__("-DABS") /* specific argument - not allowed */
```

---

To check for a specific define use:

```
#if defined(ABS)
```

If the specified option cannot be checked to determine if it is active (i.e., options that no longer exist), the message “C1439: illegal pragma `__OPTION_ACTIVE__`” is issued.

## ANSI-C Standard Types 'size\_t', 'wchar\_t' and 'ptrdiff\_t' Defines

ANSI provides some standard defines in `'stddef.h'` to deal with the implementation of defined object sizes.

Listing 6.4 show part of the contents of `stdtypes.h` (included from `stddef.h`).

#### Listing 6.4 Type Definitions of ANSI-C Standard Types

---

```
/* size_t: defines the maximum object size type */
#if defined(__SIZE_T_IS_UCHAR__)
    typedef unsigned char size_t;
#elif defined(__SIZE_T_IS_USHORT__)
    typedef unsigned short size_t;
#elif defined(__SIZE_T_IS_UINT__)
    typedef unsigned int size_t;
#elif defined(__SIZE_T_IS_ULONG__)
    typedef unsigned long size_t;
#else
```

---

## Compiler Predefined Macros

### ANSI-C Standard Types 'size\_t', 'wchar\_t' and 'ptrdiff\_t' Defines

---

```
#error "illegal size_t type"
#endif

/* ptrdiff_t: defines the maximum pointer difference type */
#if defined(__PTRDIFF_T_IS_CHAR__)
typedef signed char ptrdiff_t;
#elif defined(__PTRDIFF_T_IS_SHORT__)
typedef signed short ptrdiff_t;
#elif defined(__PTRDIFF_T_IS_INT__)
typedef signed int ptrdiff_t;
#elif defined(__PTRDIFF_T_IS_LONG__)
typedef signed long ptrdiff_t;
#else
#error "illegal ptrdiff_t type"
#endif

/* wchar_t: defines the type of wide character */
#if defined(__WCHAR_T_IS_UCHAR__)
typedef unsigned char wchar_t;
#elif defined(__WCHAR_T_IS_USHORT__)
typedef unsigned short wchar_t;
#elif defined(__WCHAR_T_IS_UINT__)
typedef unsigned int wchar_t;
#elif defined(__WCHAR_T_IS_ULONG__)
typedef unsigned long wchar_t;
#else
#error "illegal wchar_t type"
#endif
```

---

Table 6.6 lists defines that deal with other possible implementations:

**Table 6.6 Defines for Other Implementations**

Macro	Description
<code>__SIZE_T_IS_UCHAR__</code>	Defined if the Compiler expects <code>size_t</code> in <code>stddef.h</code> to be 'unsigned char'.
<code>__SIZE_T_IS_USHORT__</code>	Defined if the Compiler expects <code>size_t</code> in <code>stddef.h</code> to be 'unsigned short'.
<code>__SIZE_T_IS_UINT__</code>	Defined if the Compiler expects <code>size_t</code> in <code>stddef.h</code> to be 'unsigned int'.
<code>__SIZE_T_IS_ULONG__</code>	Defined if the Compiler expects <code>size_t</code> in <code>stddef.h</code> to be 'unsigned long'.

## Compiler Predefined Macros

ANSI-C Standard Types 'size\_t', 'wchar\_t' and 'ptrdiff\_t' Defines

**Table 6.6 Defines for Other Implementations (continued)**

Macro	Description
__WCHAR_T_IS_UCHAR__	Defined if the Compiler expects wchar_t in stddef.h to be 'unsigned char'.
__WCHAR_T_IS_USHORT__	Defined if the Compiler expects wchar_t in stddef.h to be 'unsigned short'.
__WCHAR_T_IS_UINT__	Defined if the Compiler expects wchar_t in stddef.h to be 'unsigned int'.
__WCHAR_T_IS_ULONG__	Defined if the Compiler expects wchar_t in stddef.h to be 'unsigned long'.
__PTRDIFF_T_IS_CHAR__	Defined if the Compiler expects ptrdiff_t in stddef.h to be 'char'.
__PTRDIFF_T_IS_SHORT__	Defined if the Compiler expects ptrdiff_t in stddef.h to be 'short'.
__PTRDIFF_T_IS_INT__	Defined if the Compiler expects ptrdiff_t in stddef.h to be 'int'.
__PTRDIFF_T_IS_LONG__	Defined if the Compiler expects ptrdiff_t in stddef.h to be 'long'.

The following tables show the default settings of the ANSI-C Compiler standard types size\_t and ptrdiff\_t.

## Macros for HC08

Table 6.7 lists the settings for the size\_t macro for the HC08 target.

**Table 6.7 size\_t Macro Settings**

size_t Macro	Defined
__SIZE_T_IS_UCHAR__	never
__SIZE_T_IS_USHORT__	never
__SIZE_T_IS_UINT__	always
__SIZE_T_IS_ULONG__	never

Table 6.8 lists the settings for the ptrdiff\_t macro for the HC08 target.



Table 6.8 ptrdiff\_t Macro Settings

size_t Macro	Defined
__PTRDIFF_T_IS_CHAR__	-Mt
__PTRDIFF_T_IS_SHORT__	never
__PTRDIFF_T_IS_INT__	-Ms
__PTRDIFF_T_IS_LONG__	never

## Division and Modulus

To ensure that the results of the "/" and "%" operators are defined correctly for signed arithmetic operations, both operands must be defined positive. (Refer to the backend chapter.) It is implementation-defined if the result is negative or positive when one of the operands is defined negative. This is illustrated in the Listing 6.5.

Listing 6.5 Effect of polarity upon division and modulus arithmetic

```
#ifndef __MODULO_IS_POSITIV__
  22 / 7 == 3;    22 % 7 == 1
  22 /-7 == -3;  22 % -7 == 1
-22 / 7 == -4;  -22 % 7 == 6
-22 /-7 == 4;   -22 % -7 == 6
#else
  22 / 7 == 3;    22 % 7 == +1
  22 /-7 == -3;  22 % -7 == +1
-22 / 7 == -3;  -22 % 7 == -1
-22 /-7 == 3;   -22 % -7 == -1
#endif
```

The following sections show how it is implemented in the backend.

## Macro for HC08

Table 6.9 lists the modulus macro for the HC08 target.

## Compiler Predefined Macros

### Object-File Format Defines

---

**Table 6.9 Modulus Macro Settings**

Name	Defined
<code>__MODULO_IS_POSITIV__</code>	never

## Object-File Format Defines

The Compiler defines some macros to identify the format (mainly used in the startup code if it is object file specific), depending on the specified object-file format option.

Table 6.10 lists these defines.

**Table 6.10 Object-file Format Defines**

Name	Defined
<code>__HIWARE_OBJECT_FILE_FORMAT__</code>	-Fh
<code>__ELF_OBJECT_FILE_FORMAT__</code>	-F1, -F2

## Bitfield Defines

### Bitfield Allocation

The Compiler provides six predefined macros to distinguish between the different allocations (Listing 6.6):

**Listing 6.6 Predefined bitfield-allocation macros**

---

```
__BITFIELD_MSBIT_FIRST__ /* defined if bitfield allocation
                           starts with MSBit */
__BITFIELD_LSBIT_FIRST__ /* defined if bitfield allocation
                           starts with LSBit */
__BITFIELD_MSBYTE_FIRST__ /* allocation of bytes starts with
                             MSByte */
__BITFIELD_LSBYTE_FIRST__ /* allocation of bytes starts with
                             LSByte */
__BITFIELD_MSWORD_FIRST__ /* defined if bitfield allocation
                             starts with MSWord */
__BITFIELD_LSWORD_FIRST__ /* defined if bitfield allocation
                             starts with LSWord */
```

---

Using the defines listed above, you can write compatible code over different Compiler vendors even if the bitfield allocation differs. Note that the allocation order of bitfields is important (Listing 6.7).

**Listing 6.7 Using predefined bitfield-allocation macros**

---

```
struct {
    /* Memory layout of I/O port:
        name:      MSB | CCR | DIR | DATA | DDR2
        size:      1   1   1   4   1
    */
#ifdef __BITFIELD_MSBIT_FIRST__
    unsigned int BITA:1;
    unsigned int CCR :1;
    unsigned int DIR :1;
    unsigned int DATA:4;
    unsigned int DDR2:1;
#elif defined(__BITFIELD_LSBIT_FIRST__)
    unsigned int DDR2:1;
    unsigned int DATA:4;
    unsigned int DIR :1;
    unsigned int CCR :1;
    unsigned int BITA:1;
#else
    #error "undefined bitfield allocation strategy!"
#endif
} MyIOport;
```

---

If the basic allocation unit for bitfields in the Compiler is a byte, the allocation of memory for bitfields is always from the most significant BYTE to the least significant BYTE. For example, `__BITFIELD_MSBYTE_FIRST__` is defined as shown below (Listing 6.8):

**Listing 6.8 `__BITFIELD_MSBYTE_FIRST__`**

---

```
struct {
    unsigned char a:8;
    unsigned char b:3;
    unsigned char c:5;
} MyIOport2;

/* LSBIT_FIRST      */ /* MSBIT_FIRST      */
/* MSByte  LSByte  */ /* MSByte  LSByte  */
/* aaaaaaaa cccccbbb */ /* aaaaaaaa bbbcccc  */
```

---

## Compiler Predefined Macros

### Bitfield Defines

---

**NOTE** There is no standard way to allocate bitfields. Allocation may vary from compiler to compiler even for the same target. Using bitfields for I/O register access is non-portable and inefficient for the masking involved in unpacking individual fields. It is recommended to use regular bit-and (&) and bit-or (|) operations for I/O port access.

---

## Bitfield Type Reduction

The Compiler provides two predefined macros for enabled/disabled type-size reduction (Listing 6.9). With type-size reduction enabled, the Compiler is free to reduce the type of a bitfield. For example, if the size of a bitfield is 3, the Compiler uses the char type.

### Listing 6.9 Bitfield type-reduction macros

---

```
__BITFIELD_TYPE_SIZE_REDUCTION__ /* defined if type-size
reduction is enabled */
__BITFIELD_NO_TYPE_SIZE_REDUCTION__ /* defined if type-size
reduction is disabled */
```

---

It is possible to write compatible code over different Compiler vendors and to get optimized bitfields (Listing 6.10).

### Listing 6.10 Effects of bitfield type-size reduction

---

```
struct{
    long b1:4;
    long b2:4;
} myBitfield;
31          7 3 0
-----
|#####|b2|b1| -BfaTSRoff
-----

7 3 0
-----
|b2 |b1 | -BfaTSRon
-----
```

---

## Sign of Plain Bitfields

For some architectures, the sign of a plain bitfield does not follow standard rules. Normally for the bitfield in Listing 6.11, 'myBits' is signed, because plain 'int' is also signed.

### Listing 6.11 Signed bitfield

---

```
struct _bits {  
    int myBits:3;  
} bits;
```

---

To implement it as an unsigned bitfield, use the following code in Listing 6.12:

### Listing 6.12 Unsigned bitfield

---

```
struct _bits {  
    unsigned int myBits:3;  
} bits;
```

---

However, some architectures need to overwrite this behavior to be compliant to their EABI (Embedded Application Binary Interface). Under those circumstances, the -T: Flexible Type Management (if supported) is used. The option affects the following defines (Listing 6.13):

### Listing 6.13

---

```
__PLAIN_BITFIELD_IS_SIGNED__    /* defined if plain bitfield  
                                is signed */  
__PLAIN_BITFIELD_IS_UNSIGNED__  /* defined if plain bitfield  
                                is unsigned */
```

---

## Macros for HC08

Table 6.11 identifies the implementation in the Backend for the HC08 target.

## Compiler Predefined Macros

### Bitfield Defines

---

**Table 6.11** Macros for HC08

Name	Defined
__BITFIELD_MSBIT_FIRST__	-BfaBMS
__BITFIELD_LSBIT_FIRST__	-BfaBLS
__BITFIELD_MSBYTE_FIRST__	always
__BITFIELD_LSBYTE_FIRST__	never
__BITFIELD_MSWORD_FIRST__	always
__BITFIELD_LSWORD_FIRST__	never
__BITFIELD_TYPE_SIZE_REDUCTION__	-BfaTSRon
__BITFIELD_NO_TYPE_SIZE_REDUCTION__	-BfaTSRoff
__PLAIN_BITFIELD_IS_SIGNED__	always
__PLAIN_BITFIELD_IS_UNSIGNED__	never

## Type Information Defines

The Flexible Type Management sets the defines to identify the type sizes. Table 6.12 lists these defines.

**Table 6.12** Type Information Defines

Name	Defined
__CHAR_IS_SIGNED__	see the -T option or Backend
__CHAR_IS_UNSIGNED__	see the -T option or Backend
__CHAR_IS_8BIT__	see the -T option or Backend
__CHAR_IS_16BIT__	see the -T option or Backend
__CHAR_IS_32BIT__	see the -T option or Backend
__CHAR_IS_64BIT__	see the -T option or Backend
__SHORT_IS_8BIT__	see the -T option or Backend
__SHORT_IS_16BIT__	see the -T option or Backend

**Table 6.12 Type Information Defines (continued)**

Name	Defined
__SHORT_IS_32BIT__	see the -T option or Backend
__SHORT_IS_64BIT__	see the -T option or Backend
__INT_IS_8BIT__	see the -T option or Backend
__INT_IS_16BIT__	see the -T option or Backend
__INT_IS_32BIT__	see the -T option or Backend
__INT_IS_64BIT__	see the -T option or Backend
__ENUM_IS_8BIT__	see the -T option or Backend
__ENUM_IS_SIGNED__	see the -T option or Backend
__ENUM_IS_UNSIGNED__	see the -T option or Backend
__ENUM_IS_16BIT__	see the -T option or Backend
__ENUM_IS_32BIT__	see the -T option or Backend
__ENUM_IS_64BIT__	see the -T option or Backend
__LONG_IS_8BIT__	see the -T option or Backend
__LONG_IS_16BIT__	see the -T option or Backend
__LONG_IS_32BIT__	see the -T option or Backend
__LONG_IS_64BIT__	see the -T option or Backend
__LONG_LONG_IS_8BIT__	see the -T option or Backend
__LONG_LONG_IS_16BIT__	see the -T option or Backend
__LONG_LONG_IS_32BIT__	see the -T option or Backend
__LONG_LONG_IS_64BIT__	see the -T option or Backend
__FLOAT_IS_IEEE32__	see the -T option or Backend
__FLOAT_IS_IEEE64__	see the -T option or Backend
__FLOAT_IS_DSP__	see the -T option or Backend
__DOUBLE_IS_IEEE32__	see the -T option or Backend
__DOUBLE_IS_IEEE64__	see the -T option or Backend

## Compiler Predefined Macros

### Bitfield Defines

Table 6.12 Type Information Defines (*continued*)

Name	Defined
__DOUBLE_IS_DSP__	see the -T option or Backend
__LONG_DOUBLE_IS_IEEE32__	see the -T option or Backend
__LONG_DOUBLE_IS_IEEE64__	see the -T option or Backend
__LONG_DOUBLE_IS_DSP__	see the -T option or Backend
__LONG_LONG_DOUBLE_IS_IEEE32__	see the -T option or Backend
__LONG_LONG_DOUBLE_IS_IEEE64__	see the -T option or Backend
__LONG_LONG_DOUBLE_IS_DSP__	see the -T option or Backend
__VTAB_DELTA_IS_8BIT__	see the -T option
__VTAB_DELTA_IS_16BIT__	see the -T option
__VTAB_DELTA_IS_32BIT__	see the -T option
__VTAB_DELTA_IS_64BIT__	see the -T option
__PLAIN_BITFIELD_IS_SIGNED__	see the -T option or Backend
__PLAIN_BITFIELD_IS_UNSIGNED__	see the -T option or Backend

## Freescale HC08 Specific Defines

Table 6.13 identifies specific implementations in the Backend for the HC08 target.

Table 6.13 Freescale HC08 Specific Defines

Name	Defined
__HC08__	always
__HCS08__	-Cs08
__NO_RECURSION__	never
__SMALL__	-Ms
__TINY__	-Mt
__PTR_SIZE_1__	-Mt



**Table 6.13** Freescale HC08 Specific Defines (*continued*)

<b>Name</b>	<b>Defined</b>
__PTR_SIZE_2__	-Ms
__PTR_SIZE_3__	never
__PTR_SIZE_4__	never

## Compiler Predefined Macros

*Bitfield Defines*

---

# Compiler Pragmas

---

A pragma (Listing 7.1) defines how information is passed from the Compiler Frontend to the Compiler Backend, without affecting the parser. In the Compiler, the effect of a pragma on code generation starts at the point of its definition and ends with the end of the next function. Exceptions to this rule are the pragmas `#pragma ONCE`: Include Once and `#pragma NO_STRING_CONSTR`: No String Concatenation during Preprocessing, which are valid for only one file.

## Listing 7.1 Syntax of a pragma

---

```
#pragma pragma_name [optional_arguments]
```

---

The `optional_arguments` value depends on the pragma that you use. Some pragmas do not take arguments.

---

**NOTE** A pragma directive accepts a single pragma with optional arguments. Do not place more than one pragma name in a pragma directive. The following example uses incorrect syntax:

```
#pragma ONCE NO_STRING_CONSTR
```

This is an invalid directive because two pragma names were combined into one pragma directive.

---

The following section describes all of the pragmas that affect the Frontend. All other pragmas affect only the code generation process and are described in the Backend section.

## Pragma Details

This section describes each Compiler-available pragma. The pragmas are listed in alphabetical order and are divided into separate tables. Table 7.1 lists and defines the topics that appear in the description of each pragma.

**Table 7.1 Pragma documentation topics**

<b>Topic</b>	<b>Description</b>
Scope	Scope of pragma where it is valid. (See Table 7.2, below.)
Syntax	Specifies the syntax of the pragma in an EBNF format.
Synonym	Lists a synonym for the pragma or none, if a synonym does not exist.
Arguments	Describes and lists optional and required arguments for the pragma.
Default	Shows the default setting for the pragma or none.
Description	Provides a detailed description of the pragma and how to use it.
Example	Gives an example of usage and effects of the pragma.
See also	Names related sections.

Table 7.2 is a description of the different scopes of pragmas.

**Table 7.2 Definition of items that can appear in a pragma's scope topic**

<b>Scope</b>	<b>Description</b>
File	The pragma is valid from the current position until the end of the source file. Example: If the pragma is in a header file included from a source file, the pragma is not valid in the source file.
Compilation Unit	The pragma is valid from the current position until the end of the whole compilation unit. Example: If the pragma is in a header file included from a source file, it is valid in the source file too.
Data Definition	The pragma affects only the next data definition. Ensure that you always use a data definition behind this pragma in a header file. If not, the pragma is used for the first data segment in the next header file, or in the main file.
Function Definition	The pragma affects only the next function definition. Ensure that you use this pragma in a header file: The pragma is valid for the first function in each source file where such a header file is included if there is no function definition in the header file.
Next pragma with same name	The pragma is used until the same pragma appears again. If no such pragma follows this one, it is valid until the end of the file.

## #pragma CODE\_SEG: Code Segment Definition

### Scope

Next pragma CODE\_SEG

### Syntax

```
#pragma CODE_SEG (<Modif> <Name> | DEFAULT)
```

### Synonym

CODE\_SECTION

### Arguments

<Modif>: Some of the following strings may be used:

\_\_DIRECT\_SEG (compatibility alias: DIRECT)

\_\_NEAR\_SEG (compatibility alias: NEAR)

\_\_CODE\_SEG (compatibility alias: CODE)

\_\_FAR\_SEG (compatibility alias: FAR)

The compatibility alias should not be used in new code. It only exists for backwards compatibility.

Some of the compatibility alias names do conflict with defines found in certain header files. Therefore, using them can cause hard to detect problems. Avoid using compatibility alias names.

The meaning of these segment modifiers are backend-dependent. Refer to the Freescale HC(S)08 Backend chapter for information on supported modifiers and their definitions.

<Name>: The name of the segment. This name must be used in the link parameter file on the left side of the assignment in the PLACEMENT section. Refer to the linker manual for details.

### Default

DEFAULT

### Description

This pragma specifies the function segment where it is allocated. The segment modifiers also specify the function's calling convention. The CODE\_SEG pragma sets the current code segment. This segment places all new function definitions.

## Compiler Pragmas

### Pragma Details

---

Also, all function declarations get the current code segment, when they occur. The segment modifiers of this segment determine the calling convention.

The `CODE_SEG` pragma affects function declarations as well as definitions. Ensure that all function declarations and their definitions are in the same segment.

The synonym `CODE_SECTION` has exactly the same meaning as `CODE_SEG`. See Listing 7.2 for some `CODE_SEG` examples.

#### Listing 7.2 `CODE_SEG` examples

---

```
/* in a header file */
#pragma CODE_SEG __FAR_SEG MY_CODE1
extern void f(void);
#pragma CODE_SEG MY_CODE2
extern void h(void);
#pragma CODE_SEG DEFAULT

/* in the corresponding C file: */
#pragma CODE_SEG __FAR_SEG MY_CODE1
void f(void){ /* f has FAR calling convention */
    h(); /* calls h with default calling convention */
}
#pragma CODE_SEG MY_CODE2
void h(void){ /* f has default calling convention */
    f(); /* calls f with the FAR calling convention */
}
#pragma CODE_SEG DEFAULT
```

---

---

**NOTE** Not all backends support a FAR calling convention.

---

---

**NOTE** The calling convention can also be specified with a supported keyword. The default calling convention is chosen with the memory model.

---

Listing 7.3 shows some errors when using pragmas.

#### Listing 7.3 Improper pragma usage

---

```
#pragma DATA_SEG DATA1
#pragma CODE_SEG DATA1
/* error: segment name has different types! */

#pragma CODE_SEG DATA1
#pragma CODE_SEG __FAR_SEG DATA1
/* error: segment name has modifiers! */
```

---

```
#pragma CODE_SEG DATA1
void g(void);
#pragma CODE_SEG DEFAULT
void g(void) {}
/* error: g is declared in two different segments */
#pragma CODE_SEG __FAR_SEG DEFAULT
/* error: modifiers for DEFAULT segment are not allowed */
```

---

### **See also**

- HC(S)08 Backend
- Segmentation
- Linker Manual
- `#pragma CONST_SEG`: Constant Data Segment Definition
- `#pragma DATA_SEG`: Data Segment Definition
- `#pragma STRING_SEG`: String Segment Definition
- `-Cc`: Allocate Constant Objects into ROM compiler option

## #pragma CONST\_SEG: Constant Data Segment Definition

### Scope

Next pragma CONST\_SEG

### Syntax

```
#pragma CONST_SEG (<Modif> <Name> | DEFAULT)
```

### Synonym

CONST\_SECTION

### Arguments

<Modif>: Some of the following strings may be used:

- \_\_SHORT\_SEG (compatibility alias: SHORT)
- \_\_DIRECT\_SEG (compatibility alias: DIRECT)
- \_\_NEAR\_SEG (compatibility alias: NEAR)
- \_\_CODE\_SEG (compatibility alias: CODE)
- \_\_FAR\_SEG (compatibility alias: FAR)

---

**NOTE** A compatibility alias should not be used in new code. It only exists for backwards compatibility. Some of the compatibility alias names conflict with defines found in certain header files. Therefore, using them can cause hard to detect problems. Avoid using compatibility alias names.

---

The segment modifiers are backend-dependent. Refer to the Freescale HC(S)08 Backend chapter to find the supported modifiers and their meanings. The \_\_SHORT\_SEG modifier specifies a segment which is accessed with - bit addresses.

<Name>: The name of the segment. This name must be used in the link parameter file on the left side of the assignment in the PLACEMENT part. Please refer to the linker manual for details.

### Default

DEFAULT



## Description

This pragma allocates constant variables into a segment. The segment is then located in the link parameter file to specific addresses.

The pragma `CONST_SEG` sets the current const segment. This segment places all constant variable declarations. The default segment is set with:

```
#pragma CONST_SEG DEFAULT
```

Constants are allocated in the current data segment that is defined with `#pragma DATA_SEG`: Data Segment Definition in the HIWARE object-file format when the `-Cc`: Allocate Constant Objects into ROM option is not specified and until the first `#pragma CONST_SEG` occurs in the source. With the `-Cc` option, constants are always allocated in constant segments in the ELF object-file format and after the first `#pragma CONST_SEG`.

The `CONST_SEG` pragma also affects constant variable declarations as well as definitions. Ensure that all constant variable declarations and definitions are in the same const segment.

Some compiler optimizations assume that objects having the same segment are placed together. Backends supporting banked data, for example, may set the page register only once for two accesses to two different variables in the same segment. This is also the case for the `DEFAULT` segment. When using a paged access to variables, place one segment on one page in the link parameter file.

When `#pragma INTO_ROM`: Put Next Variable Definition into ROM is active, the current const segment is not used.

The synonym `CONST_SECTION` has exactly the same meaning as `CONST_SEG`.

## Examples

Listing 7.4 shows examples of the `CONST_SEG` pragma.

### Listing 7.4 Examples of the `CONST_SEG` pragma

---

```
/* Use the pragmas in a header file */
#pragma CONST_SEG __SHORT_SEG SHORT_CONST_MEMORY
extern const int i_short;
#pragma CONST_SEG CUSTOM_CONST_MEMORY
extern const int j_custom;
#pragma CONST_SEG DEFAULT

/* Some C file, which includes the above header file code */

void main(void) {
    int k= i; /* may use short access */
    k= j;
```

## Compiler Pragmas

### Pragma Details

---

```
}

/* in the C file defining the constants : */
#pragma CONST_SEG __SHORT_SEG SHORT_CONST_MEMORY
extern const int i_short=7
#pragma CONST_SEG CUSTOM_CONST_MEMORY
extern const int j_custom=8;
#pragma CONST_SEG DEFAULT
```

---

Listing 7.5 shows code that uses the `CONST_SEG` pragma *improperly*.

#### Listing 7.5 Improper use of the `CONST_SEG` pragma

---

```
#pragma DATA_SEG CONST1
#pragma CONST_SEG CONST1 /* error: segment name has different types!*/

#pragma CONST_SEG C2
#pragma CONST_SEG __SHORT_SEG C2 // error: segment name has modifiers!

#pragma CONST_SEG CONST1
extern int i;
#pragma CONST_SEG DEFAULT
int i; /* error: i is declared in two different segments */

#pragma CONST_SEG __SHORT_SEG DEFAULT // error: modifiers for DEFAULT
// segment are not allowed
```

---

#### See also

- HC(S)08 Backend
- Segmentation
- Linker Manual
- `#pragma CODE_SEG`: Code Segment Definition
- `#pragma DATA_SEG`: Data Segment Definition
- `#pragma STRING_SEG`: String Segment Definition
- Cc: Allocate Constant Objects into ROM compiler option
- `#pragma INTO_ROM`: Put Next Variable Definition into ROM

## #pragma CREATE\_ASM\_LISTING: Create an Assembler Include File Listing

### Scope

Next pragma CREATE\_ASM\_LISTING

### Syntax

```
#pragma CREATE_ASM_LISTING (ON|OFF)
```

### Synonym

None

### Arguments

ON: All following defines or objects are generated

OFF: All following defines or objects are not generated

### Default

OFF

### Description

This pragma controls if the following defines or objects are printed into the assembler include file.

A new file is only generated when the `-La` compiler option is specified together with a header file containing `"#pragma CREATE_ASM_LISTING ON"`.

### Listing 7.6 Example

---

```
#pragma CREATE_ASM_LISTING ON
extern int i; /* i is accessible from the asm code */

#pragma CREATE_ASM_LISTING OFF
extern int j; /* j is only accessible from the C code */
```

---

### See also

`-La`: Generate Assembler Include File  
Generating Assembler Include Files (`-La` compiler option)

## #pragma DATA\_SEG: Data Segment Definition

### Scope

Next pragma DATA\_SEG

### Syntax

```
#pragma DATA_SEG (<Modif> <Name> | DEFAULT)
```

### Synonym

DATA\_SECTION

### Arguments

<Modif>: Some of the following strings may be used:

\_\_SHORT\_SEG (compatibility alias: SHORT)

\_\_DIRECT\_SEG (compatibility alias: DIRECT)

\_\_NEAR\_SEG (compatibility alias: NEAR)

\_\_CODE\_SEG (compatibility alias: CODE)

\_\_FAR\_SEG (compatibility alias: FAR)

A compatibility alias should not be used in new code. It only exists for backwards compatibility.

Some of the compatibility alias names do conflict with defines found in certain header files. Therefore, using them can cause hard to detect problems. Avoid using compatibility alias names.

The \_\_SHORT\_SEG modifier specifies a segment which is accessed with 8-bit addresses. The meaning of these segment modifiers are backend-dependent. Read the backend chapter to find the supported modifiers and their meaning.

<Name>: The name of the segment. This name must be used in the link parameter file on the left side of the assignment in the PLACEMENT part. Please refer to the linker manual for details.

### Default

DEFAULT

## Description

This pragma allocates variables into a segment. This segment is then located in the link parameter file to specific addresses.

The `DATA_SEG` pragma sets the current data segment. This segment is used to place all variable declarations. The default segment is set with:

```
#pragma DATA_SEG DEFAULT
```

Constants are also allocated in the current data segment in the HIWARE object-file format when the option `-cc` is not specified and no “`#pragma CONST_SEG`” occurred in the source. When using the `-Cc: Allocate Constant Objects into ROM` compiler option and the ELF object-file format, constants are not allocated in the data segment.

The `DATA_SEG` pragma also affects data declarations, as well as definitions. Ensure that all variable declarations and definitions are in the same segment.

Some compiler optimizations assume that objects having the same segment are together. Backends supporting banked data, for example, may set the page register only once if two accesses two different variables in the same segment are done. This is also the case for the `DEFAULT` segment. When using a paged access to constant variables, put one segment on one page in the link parameter file.

When `#pragma INTO_ROM: Put Next Variable Definition into ROM` is active, the current data segment is not used.

The `DATA_SECTION` synonym has exactly the same meaning as `DATA_SEG`.

## Example

Listing 7.7 is an example using the `DATA_SEG` pragma.

### Listing 7.7 Using the `DATA_SEG` pragma

---

```
/* in a header file */
#pragma DATA_SEG __SHORT_SEG SHORT_MEMORY
extern int i_short;
#pragma DATA_SEG CUSTOM_MEMORY
extern int j_custom;
#pragma DATA_SEG DEFAULT

/* in the corresponding C file : */
#pragma DATA_SEG __SHORT_SEG SHORT_MEMORY
int i_short;
#pragma DATA_SEG CUSTOM_MEMORY
int j_custom;
#pragma DATA_SEG DEFAULT

void main(void) {
```

## Compiler Pragmas

### Pragma Details

---

```
i = 1; /* may use short access */
j = 5;
}
```

---

Listing 7.8 shows code that uses the `DATA_SEG` pragma *improperly*.

#### Listing 7.8 Improper use of the `DATA_SEG` pragma

---

```
#pragma DATA_SEG DATA1
#pragma CONST_SEG DATA1 /* error: segment name has different types! */

#pragma DATA_SEG DATA1
#pragma DATA_SEG __SHORT_SEG DATA1
/* error: segment name has modifiers! */

#pragma DATA_SEG DATA1
extern int i;
#pragma DATA_SEG DEFAULT
int i; /* error: i is declared in different segments */

#pragma DATA_SEG __SHORT_SEG DEFAULT
/* error: modifiers for DEFAULT segment are not allowed */
```

---

#### See also

- HC(S)08 Backend Segmentation
- Linker section of the Build Tool Utilities manual
- `#pragma CODE_SEG`: Code Segment Definition
- `#pragma CONST_SEG`: Constant Data Segment Definition
- `#pragma STRING_SEG`: String Segment Definition
- Cc: Allocate Constant Objects into ROM compiler option
- `#pragma INTO_ROM`: Put Next Variable Definition into ROM

## #pragma INLINE: Inline Next Function Definition

### Scope

Function Definition

### Syntax

```
#pragma INLINE
```

### Synonym

None

### Arguments

None

### Default

None

### Description

This pragma directs the Compiler to inline the next function in the source.

The pragma is the same as using the `-Oi` compiler option.

### Listing 7.9 Using an INLINE pragma to inline a function

---

```
int i;
#pragma INLINE
static void foo(void) {
    i = 12;
}
void main(void) {
    foo(); // results in inlining 'i = 12;'
}
```

---

### See also

`#pragma NO_INLINE`: Do not Inline Next Function Definition  
`-Oi`: Inlining compiler option

## **#pragma INTO\_ROM: Put Next Variable Definition into ROM**

### **Scope**

Data Definition

### **Syntax**

```
#pragma INTO_ROM
```

### **Synonym**

None

### **Arguments**

None

### **Default**

None

### **Description**

This pragma forces the next (non-constant) variable definition to be `const` (together with the `-Cc` compiler option).

The pragma is active only for the next single variable definition. A following segment pragma (`CONST_SEG`, `DATA_SEG`, `CODE_SEG`) disables the pragma.

---

**NOTE** This pragma is only useful for the HIWARE object-file format (and not for ELF/DWARF).

---

---

**NOTE** This pragma is to force a non-constant (means normal ‘variable’) object to be recognized as ‘`const`’ by the compiler. If the variable already is declared as ‘`const`’ in the source, this pragma is not needed. This pragma was introduced to cheat the constant handling of the compiler, and shall not be used any longer. It is supported for legacy reasons only.

---



### **Example**

Listing 7.10 is an example using the INTO\_ROM pragma.

#### **Listing 7.10 Using the INTO\_ROM pragma**

---

```
#pragma INTO_ROM
char *const B[] = {"hello", "world"};

#pragma INTO_ROM
int constVariable; /* put into ROM_VAR, .rodata */

int other; /* put into default segment */

#pragma INTO_ROM
#pragma DATA_SEG MySeg /* INTO_ROM overwritten! */
int other2; /* put into MySeg */
```

---

### **See also**

-Cc: Allocate Constant Objects into ROM compiler option

## #pragma LINK\_INFO: Pass Information to the Linker

### Scope

Function

### Syntax

```
#pragma LINK_INFO NAME "CONTENT"
```

### Synonym

None

### Arguments

NAME: Identifier specific to the purpose of this LINK\_INFO.

CONTENT: C style string containing only printable ASCII characters.

### Default

None

### Description

This pragma instructs the compiler to put the passed name content pair into the ELF file. For the compiler, the used name and its content do have no meaning other than one name can only contain one content. However, multiple pragmas with different NAMES are legal.

For the linker or for the debugger however, the NAME might trigger some special functionality with CONTENT as an argument.

The linker collects the CONTENT for every NAME in different object files and issues an message if a different CONTENT is given for different object files.

---

**NOTE** This pragma only works with the ELF object-file format.

---

### Example

Apart from extended functionality implemented in the linker or debugger, this feature can also be used for user-defined link-time consistency checks:

Using the code shown in Listing 7.11 in a header file used by all compilation units, the linker will issue a message if the object files built with `_DEBUG` are linked with object files built without it.

**Listing 7.11 Using pragmas to assist in debugging**

---

```
#ifdef _DEBUG
#pragma LINK_INFO MY_BUILD_ENV DEBUG
#else
#pragma LINK_INFO MY_BUILD_ENV NO_DEBUG
#endif
```

---

## Compiler Pragmas

### Pragma Details

---

## #pragma LOOP\_UNROLL: Force Loop Unrolling

### Scope

Function

### Syntax

```
#pragma LOOP_UNROLL
```

### Synonym

None

### Arguments

None

### Default

None

### Description

If this pragma is present, loop unrolling is performed for the next function. This is the same as if the `-Cu` option is set for the following single function (Listing 7.12).

### Listing 7.12 Using the LOOP\_UNROLL pragma to unroll a for loop

---

```
#pragma LOOP_UNROLL
void F(void) {
    for (i=0; i<5; i++) { // unrolling this loop
        ...
    }
}
```

---

### See also

`#pragma NO_LOOP_UNROLL`: Disable Loop Unrolling  
`-Cu`: Loop Unrolling

## #pragma mark: Entry in CodeWarrior IDE Function List

### Scope

Line

### Syntax

```
#pragma mark {any text}
```

### Synonym

None

### Arguments

None

### Default

None

### Description

This pragma adds an entry into the function list of the CodeWarrior IDE. It also helps to introduce faster code lookups by providing a menu entry which directly jumps to a code position. With the special “#pragma mark -”, a separator line is inserted.

---

**NOTE** The compiler does not actually handle this pragma. The compiler ignores this pragma. The CodeWarrior IDE scans opened source files for this pragma. It is not necessary to recompile a file when this pragma is changed. The IDE updates its menus instantly.

---

### Example

In the example in Listing 7.13, the pragma accesses declarations and definitions.

#### Listing 7.13 Using the MARK pragma

---

```
#pragma mark local function declarations
static void inc_counter(void);
static void inc_ref(void);

#pragma mark local variable definitions
static int counter;
```

---

## Compiler Pragmas

### *Pragma Details*

---

```
static int ref;

#pragma mark -
static void inc_counter(void) {
    counter++;
}
static void inc_ref(void) {
    ref++;
}
```

---

## #pragma MESSAGE: Message Setting

### Scope

Compilation Unit or until the next MESSAGE pragma

### Syntax

```
#pragma MESSAGE { (WARNING | ERROR |  
INFORMATION | DISABLE | DEFAULT) { <CNUM> } }
```

### Synonym

None

### Arguments

<CNUM>: Number of message to be set in the C1234 format

### Default

None

### Description

Messages are selectively set to an information message, a warning message, a disable message, or an error message.

---

**NOTE** This pragma has no effect for messages which are produced during preprocessing. The reason is that the pragma parsing has to be done during normal source parsing but not during preprocessing.

---

---

**NOTE** This pragma (as other pragmas) has to be specified outside of the function scope. e.g., it is not possible to change a message inside a function or for a part of a function.

---

### Example

In the example shown in Listing 7.14, parentheses ( ) were left out.

#### Listing 7.14 Using the MESSAGE Pragma

---

```
/* treat C1412: Not a function call, */  
/* address of a function, as error */  
#pragma MESSAGE ERROR C1412
```

---

## Compiler Pragmas

### *Pragma Details*

---

```
void f(void);
void main(void) {
    f; /* () is missing, but still legal in C */
    /* ERROR because of pragma MESSAGE */
}
```

---

### **See also**

- WmsgSd: Setting a Message to Disable
- WmsgSe: Setting a Message to Error
- WmsgSi: Setting a Message to Information
- WmsgSw: Setting a Message to Warning



## #pragma NO\_ENTRY: No Entry Code

### Scope

Function

### Syntax

```
#pragma NO_ENTRY
```

### Synonym

None

### Arguments

None

### Default

None

### Description

This pragma suppresses the generation of the entry code and is useful for inline assembler functions.

The code generated in a function with `#pragma NO_ENTRY` may not be safe. It is assumed that the user ensures stack use.

---

**NOTE** Not all backends support this pragma. Some still generate entry code even if this pragma is specified.

---

### Example

Listing 7.15 shows how to use the `NO_ENTRY` pragma (along with others) to avoid any generated code by the compiler. All code is written in inline assembler.

#### Listing 7.15 Blocking compiler-generated function-management instructions

---

```
#pragma NO_ENTRY
#pragma NO_EXIT
#pragma NO_FRAME
#pragma NO_RETURN
void Func0(void) {
    __asm { /* no code should be written by the compiler.*/
```

## Compiler Pragmas

### *Pragma Details*

---

```
    ...  
  }  
}
```

---

#### **See also**

`#pragma NO_EXIT`: No Exit Code

`#pragma NO_FRAME`: No Frame Code

`#pragma NO_RETURN`: No Return Instruction

## #pragma NO\_EXIT: No Exit Code

### Scope

Function

### Syntax

```
#pragma NO_EXIT
```

### Synonym

None

### Arguments

None

### Default

None

### Description

This pragma suppresses generation of the exit code and is useful for inline assembler functions.

The code generated in a function with #pragma NO\_ENTRY may not be safe. It is assumed that the user ensures stack usage.

---

**NOTE** Not all backends support this pragma. Some still generate exit code even if this pragma is specified.

---

### Example

Listing 7.16 shows how to use the NO\_EXIT pragma (along with others) to avoid any generated code by the compiler. All code is written in inline assembler.

#### Listing 7.16 Blocking Compiler-generated function management instructions

---

```
#pragma NO_ENTRY
#pragma NO_EXIT
#pragma NO_FRAME
#pragma NO_RETURN
void Func0(void) {
    __asm { /* no code should be written by the compiler.*/
```

## Compiler Pragmas

### *Pragma Details*

---

```
    ...  
  }  
}
```

---

#### **See also**

#pragma NO\_ENTRY: No Entry Code

#pragma NO\_FRAME: No Frame Code

#pragma NO\_RETURN: No Return Instruction

## #pragma NO\_FRAME: No Frame Code

### Scope

Function

### Syntax

```
#pragma NO_FRAME
```

### Synonym

None

### Arguments

None

### Default

None

### Description

This pragma suppresses the generation of frame code and is useful for inline assembler functions.

The code generated in a function with #pragma NO\_ENTRY may not be safe. It is assumed that the user ensures stack usage.

---

**NOTE** Not all backends support this pragma. Some still generate frame code even if this pragma is specified.

---

### Example

Listing 7.17 shows how to use the NO\_FRAME pragma (along with others) to avoid any generated code by the compiler. All code is written in inline assembler.

#### Listing 7.17 Blocking compiler-generated function management instructions

---

```
#pragma NO_ENTRY
#pragma NO_EXIT
#pragma NO_FRAME
#pragma NO_RETURN
void Func0(void) {
    __asm { /* no code should be written by the compiler.*/
```

## Compiler Pragmas

### *Pragma Details*

---

```
    ...  
  }  
}
```

---

#### **See also**

`#pragma NO_ENTRY`: No Entry Code

`#pragma NO_EXIT`: No Exit Code

`#pragma NO_RETURN`: No Return Instruction

## #pragma NO\_INLINE: Do not Inline Next Function Definition

### Scope

Function

### Syntax

```
#pragma NO_INLINE
```

### Synonym

None

### Arguments

None

### Default

None

### Description

This pragma prevents the Compiler to inline the next function in the source. The pragma is used to avoid to inline a function which would be otherwise inlined because of the `-Oi` compiler option.

### Listing 7.18 Use of #pragma NO\_INLINE to prevent inlining a function.

---

```
// (With option -Oi)
int i;
#pragma NO_INLINE
static void foo(void) {
    i = 12;
}

void main(void) {
    foo(); // call is not inlined
}
```

---

### See also

`#pragma INLINE`: Inline Next Function Definition  
`-Oi`: Inlining compiler option

### #pragma NO\_LOOP\_UNROLL: Disable Loop Unrolling

#### Scope

Function

#### Syntax

```
#pragma NO_LOOP_UNROLL
```

#### Synonym

None

#### Arguments

None

#### Default

None

#### Description

If this pragma is present, no loop unrolling is performed for the next function definition (Listing 7.19), even if the `-Cu` command line option is given.

#### Listing 7.19 Using the NO\_LOOP\_UNROLL pragma to temporarily halt loop unrolling

---

```
#pragma NO_LOOP_UNROLL
void F(void) {
    for (i=0; i<5; i++) { // loop is NOT unrolled
        ...
    }
}
```

---

#### See also

`#pragma LOOP_UNROLL`: Force Loop Unrolling  
`-Oi`: Inlining compiler option



## #pragma NO\_RETURN: No Return Instruction

### Scope

Function

### Syntax

```
#pragma NO_RETURN
```

### Synonym

### Arguments

### Default

### Description

This pragma suppresses the generation of the return instruction (return from subroutine or return from interrupt). This may be useful if you care about the return instruction itself or if the code has to fall through to the first instruction of the next function.

This pragma does not suppress the generation of the exit code at all (e.g., deallocation of local variables or compiler generated local variables). The pragma suppresses the generation of the return instruction.

---

**NOTE** If this feature is used to fall through to the next function, smart linking has to be switched off in the Linker, because the next function may be not referenced from somewhere else. In addition, be careful that both functions are in a linear segment. To be on the safe side, allocate both function into a segment that only has a linear memory area.

---

### Example

The example in Listing 7.20 places some functions into a special named segment. All functions in this special code segment have to be called from an operating system every 2 seconds after each other. With the pragma, some functions do not return. They fall directly to the next function to be called, saving code size and execution time.

## Compiler Pragmas

### *Pragma Details*

---

#### **Listing 7.20 Blocking compiler-generated function return instructions**

---

```
#pragma CODE_SEG CallEvery2Secs
#pragma NO_RETURN
void Func0(void) {
    /* first function, called from OS */
    ...
} /* fall through!!!! */
#pragma NO_RETURN
void Func1(void) {
    ...
} /* fall through */
...
/* last function has to return, no pragma is used! */
void FuncLast(void) {
    ...
}
```

---

#### **See also**

`#pragma NO_ENTRY`: No Entry Code

`#pragma NO_EXIT`: No Exit Code

`#pragma NO_FRAME`: No Frame Code

## #pragma NO\_STRING\_CONSTR: No String Concatenation during Preprocessing

### Scope

Compilation Unit

### Syntax

```
#pragma NO_STRING_CONSTR
```

### Synonym

### Arguments

### Default

### Description

This pragma is valid for the rest of the file in which it appears. It switches off the special handling of '#' as a string constructor. This is useful if a macro contains inline assembler statements using this character, e.g., for IMMEDIATE values.

### Example

The following pseudo assembly-code macro (Listing 7.21) shows the use of the pragma. Without the pragma, '#' is handled as a string constructor, which is not the desired behavior.

#### Listing 7.21 Using a NO\_STRING\_CONSTR pragma in order to alter the meaning of #

---

```
#pragma NO_STRING_CONSTR
#define HALT(x)    __asm { \
                    LOAD Reg, #3 \
                    HALT x, #255\
                }
```

---

### See also

Using the Immediate-Addressing Mode in HLI Assembler Macros

## **#pragma ONCE: Include Once**

### **Scope**

File

### **Syntax**

```
#pragma ONCE
```

### **Synonym**

None

### **Arguments**

None

### **Default**

None

### **Description**

If this pragma appears in a header file, the file is opened and read only once. This increases compilation speed.

### **Example**

```
#pragma ONCE
```

### **See also**

-Pio: Include Files Only Once compiler option

## #pragma OPTION: Additional Options

### Scope

Compilation Unit or until next pragma OPTION

### Syntax

---

```
#pragma OPTION (ADD [<Handle>]{<Option>}|DEL ({Handle}|ALL))
```

---

### Synonym

None

### Arguments

<Handle>: An identifier - added options can selectively be deleted.

<Option>: A valid option string enclosed in double quote ( " ) characters

### Default

None

### Description

Options are added inside of the source code while compiling a file.

The options given on the command line or in a configuration file cannot be changed in any way.

Additional options are added to the current ones with the “ADD” command. A handle may be given optionally.

The “DEL” command either removes all options with a specific handle. It also uses the “ALL” keyword to remove all added options regardless if they have a handle or not. Note that you only can remove options which were added previously with the `pragma OPTION ADD`.

All keywords and the handle are case-sensitive.

Restrictions:

- The `-D`: Macro Definition (preprocessor definition) compiler option is not allowed. Use a `#define` preprocessor directive instead.
- The `-OdocF`: Dynamic Option Configuration for Functions compiler option is not allowed. Specify this option on the command line or in a configuration file instead.

## Compiler Pragmas

### Pragma Details

---

- The Message Setting compiler options have no effect:
  - -WmsgSd: Setting a Message to Disable
  - -WmsgSe: Setting a Message to Error
  - -WmsgSi: Setting a Message to Information
  - -WmsgSw: Setting a Message to WarningUse #pragma MESSAGE: Message Setting instead.
- Only options concerning tasks during code generation are used. Options controlling the preprocessor, for example, have no effect.
- No macros are defined for specific options.
- Only options having function scope may be used.
- The given options must not specify a conflict to any other given option.
- The pragma is not allowed inside of declarations or definitions.

### Example

The example in Listing 7.22 shows how to compile only a single function with the additional -Or option.

#### Listing 7.22 Using the OPTION Pragma

---

```
#pragma OPTION ADD function_main_handle "-Or"

int sum(int max) { /* compiled with -or */
    int i, sum=0;
    for (i = 0; i < max; i++) {
        sum += i;
    }
    return sum;
}

#pragma OPTION DEL function_main_handle
/* now the same options as before the #pragma */
/* OPTION ADD are active again */
```

---

The examples in Listing 7.23 show *improper* uses of the OPTION pragma.

#### Listing 7.23 Improper uses of the OPTION pragma

---

```
#pragma OPTION ADD -Or /* ERROR, use "-Or" */
#pragma OPTION "-Or" /* ERROR, use keyword ADD */
#pragma OPTION ADD "-Odocf=\"-Or\""
/* ERROR, "-Odocf" not allowed in this pragma */
```

---

```
void f(void) {  
#pragma OPTION ADD "-Or"  
/* ERROR, pragma not allowed inside of declarations */  
}  
#pragma OPTION ADD "-Cni"  
#ifndef __CNI__  
/* ERROR, macros are not defined for options */  
/* added with the pragma */  
#endif
```

---

## #pragma STRING\_SEG: String Segment Definition

### Scope

Next pragma STRING\_SEG

### Syntax

```
#pragma STRING_SEG (<Modif><Name> | DEFAULT)
```

### Synonym

STRING\_SECTION

### Arguments

<Modif>: Some of the following strings may be used:

\_\_DIRECT\_SEG (compatibility alias: DIRECT)

\_\_NEAR\_SEG (compatibility alias: NEAR)

\_\_CODE\_SEG (compatibility alias: CODE)

\_\_FAR\_SEG (compatibility alias: FAR)

---

**NOTE** A compatibility alias should not be used in new code. It only exists for backwards compatibility.  
Some of the compatibility alias names conflict with defines found in certain header files. Avoid using compatibility alias names.

---

The \_\_SHORT\_SEG modifier specifies a segment that accesses using 8-bit addresses. The definitions of these segment modifiers are backend-dependent. Read the backend chapter to find the supported modifiers and their definitions.

<Name>: The name of the segment. This name must be used in the link parameter file on the left side of the assignment in the PLACEMENT part. Please refer to the linker manual for details.

### Default

DEFAULT



### Description

This pragma allocates strings into a segment. Strings are allocated in the linker segment `STRINGS`. This pragma allocates strings in special segments. String segments also may have modifiers. This instructs the Compiler to access them in a special way when necessary.

Segments defined with the pragma `STRING_SEG` are treated by the linker like constant segments defined with `#pragma CONST_SEG`, so they are allocated in ROM areas.

The pragma `STRING_SEG` sets the current string segment. This segment is used to place all newly occurring strings.

---

**NOTE** The linker may support a overlapping allocation of strings. e.g., the allocation of “CDE” inside of the string “ABCDE”, so that both strings together need only six bytes. When putting strings into user-defined segments, the linker may no longer do this optimization. Only use a user-defined string segment when necessary.

---

The synonym `STRING_SECTION` has exactly the same meaning as `STRING_SEG`.

### Example

Listing 7.24 is an example of the `STRING_SEG` pragma allocating strings into a segment with the name, `STRING_MEMORY`.

#### Listing 7.24 Using a `STRING_SEG` pragma to allocate a segment for strings

---

```
#pragma STRING_SEG STRING_MEMORY
char* p="String1";
void f(char*);
void main(void) {
    f("String2");
}
#pragma STRING_SEG DEFAULT
```

---

### See also

- HC(S)08 Backend Segmentation
- Linker section of the Build Tool Utilities manual
- `#pragma CODE_SEG`: Code Segment Definition
- `#pragma CONST_SEG`: Constant Data Segment Definition
- `#pragma DATA_SEG`: Data Segment Definition

## #pragma TEST\_CODE: Check Generated Code

### Scope

Function Definition

### Syntax

```
#pragma TEST_CODE CompOp <Size> {<HashCode>}  
CompOp: == | != | < | > | <= | >=
```

### Arguments

<Size>: Size of the function to be used with compare operation  
<HashCode>: optional value specifying one specific code pattern.

### Default

None

### Description

This pragma checks the generated code. If the check fails, the message C3601 is issued.

The following parts are tested:

- Size of the function:

The compare operator and the size given as arguments are compared with the size of the function.

This feature checks that the compiler generates less code than a given boundary. Or, to be sure that certain code it can also be checked that the compiler produces more code than specified. To only check the hashcode, use a condition which is always TRUE, such as "`!= 0`".

- Hashcode:

The compiler produces a 16-bit hashcode from the produced code of the next function. This hashcode considers:

- The code bytes of the generated functions
- The type, offset, and addend of any fixup.

To get the hashcode of a certain function, compile the function with an active #pragma TEST\_CODE which will fail. Then copy the computed hashcode out of the body of the message C3601.

---

**NOTE** The code generated by the compiler may change. If the test fails, it is often not certain that the topic chosen to be checked was wrong.

---

### Examples

Listing 7.25 and Listing 7.26 present two examples of the `TEST_CODE` pragma.

#### Listing 7.25 Using `TEST_CODE` to check the size of generated object code

---

```
/* check that an empty function is smaller */
/* than 10 bytes */
#pragma TEST_CODE < 10
void main(void) {
}
```

---

You can also use the `TEST_CODE` pragma to detect when a different code is generated (Listing 7.26).

#### Listing 7.26 Using a `Test_Code` pragma with a hashcode

---

```
/* If the following pragma fails, check the code. */
/* If the code is OK, add the hashcode to the */
/* list of allowed codes : */
#pragma TEST_CODE != 0 25645 37594
/* check code patterns : */
/* 25645 : shift for *2 */
/* 37594 : mult for *2 */
void main(void) {
    f(2*i);
}
```

---

### See also

Message C3601

## **#pragma TRAP\_PROC: Mark function as interrupt Function**

### **Scope**

Function Definition

### **Syntax**

```
#pragma TRAP_PROC
```

### **Arguments**

See Backend

### **Default**

None

### **Description**

This pragma marks a function to be an interrupt function. Because interrupt functions may need some special entry and exit code, this pragma has to be used for interrupt functions.

Do not use this pragma for declarations (e.g., in header files), as the pragma is valid for the next definition.

See the Backend chapter for details.

### **Example**

Listing 7.27 marks the `MyInterrupt()` function as an interrupt function.

#### **Listing 7.27 Using the TRAP\_PROC pragma to mark an interrupt function**

---

```
#pragma TRAP_PROC
void MyInterrupt(void) {
    ...
}
```

---

### **See also**

interrupt Keyword

# ANSI-C Frontend

---

The Compiler Frontend reads the source files, does all the syntactic and semantic checking, and produces intermediate representation of the program which then is passed on to the Backend to generate code.

This section discusses features, restrictions, and further properties of the ANSI-C Compiler Frontend.

## Implementation Features

The Compiler provides a series of pragmas instead of introducing additions to the language to support features such as interrupt procedures. The Compiler implements ANSI-C according to the X3J11 standard. The reference document is “*American National Standard for Programming Languages – C*”, ANSI/ISO 9899–1990.

## Keywords

See Listing 8.1 for the complete list of ANCSI-C keywords.

**Listing 8.1 ANSI-C keywords**

---

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while

---

## Preprocessor Directives

The Compiler supports the full set of preprocessor directives as required by the ANSI standard (Listing 8.2).

**Listing 8.2 ANSI-C preprocessor directives.**

---

```
#if, #ifdef, #ifndef, #else, #elif, #endif  
#define, #undef  
#include  
#pragma  
#error, #line
```

---

The preprocessor operators `defined`, `#`, and `##` are also supported. There is a special non-ANSI directive `#warning` which is the same as `#error`, but issues only a warning message.

## Language Extensions

There is a language extension in the Compiler for ANSI-C. You can use keywords to qualify pointers in order to distinguish them, or to mark interrupt routines.

The Compiler supports the following non-ANSI compliant keywords (see Backend if they are supported and for their semantics):

## Pointer Qualifiers

Pointer qualifiers (Listing 8.3) can be used to distinguish between different pointer types (e.g., for paging). Some of them are also used to specify the calling convention to be used (e.g., if banking is available).

**Listing 8.3 Pointer qualifiers**

---

```
__far (alias far)  
__near (alias near)
```

---

To allow portable programming between different CPUs (or if the target CPU does not support an additional keyword), you can include the defines listed below in the `'hidef.h'` header file (Listing 8.4).

**Listing 8.4 far and near can be defined in the hidef.h file**

---

```
#define far /* no far keyword supported */  
#define near /* no near keyword supported */
```

---

## Special Keywords

ANSI-C was not designed with embedded controllers in mind. The listed keywords (Listing 8.5) do not conform to ANSI standards. However, they do enable an easy way to achieve good results from code used for embedded applications.

### Listing 8.5 Special (non-ANSI) keywords

---

```
__alignof__  
__va_sizeof__  
__interrupt (alias interrupt)  
__asm (aliases _asm and asm)
```

---

**NOTE** See the *Non-ANSI Keywords* section in the HC(S)08 Backend for more details. You can use the `__interrupt` keyword to mark functions as interrupt functions, and to link the function to a specified interrupt vector number (not supported by all backends).

---

## Binary Constants (0b)

It is as well possible to use the binary notation for constants instead of hexadecimal constants or normal constants. Note that binary constants are not allowed if the `-Ansi:Strict` ANSI compiler option is switched on. Binary constants start with the `0b` prefix, followed by a sequence of 0 or 1.

### Listing 8.6 Demonstration of a Binary Constant

---

```
#define myBinaryConst 0b01011  
int i;  
  
void main(void) {  
    i = myBinaryConst;  
}
```

---

## Hexadecimal Constants (\$)

It is possible to use Hexadecimal constants inside HLI (High Level Inline) Assembly. For example, instead of `0x1234` you can use `$1234`. Note that this is valid only for inline assembly.

## **#warning Directive**

The `#warning` directive (Listing 8.7) is used as it is similar to the `#error` directive.

### **Listing 8.7 #warning directive.**

---

```
#ifndef MY_MACRO

    #warning "MY_MACRO set to default"
    #define MY_MACRO 1234
#endif
```

---

## **Global Variable Address Modifier (@address)**

You can assign global variables to specific addresses with the global variable address modifier. These variables are called ‘absolute variables’. They are useful for accessing memory mapped I/O ports and have the following syntax:

---

```
Declaration = <TypeSpec> <Declarator>[@<Address>|@"<Section>"]
              [= <Initializer>];
```

---

<TypeSpec> is the type specifier, e.g., `int`, `char`

<Declarator> is the identifier of the global object, e.g., `i`, `glob`

<Address> is the absolute address of the object, e.g., `0xff04`, `0x00+8`

<Initializer> is the value to which the global variable is initialized.

A segment is created for each global object specified with an absolute address. This address must not be inside any address range in the `SECTIONS` entries of the link parameter file. Otherwise, there would be a linker error (overlapping segments). If the specified address has a size greater than that used for addressing the default data page, pointers pointing to this global variable must be `"__far"`. An alternate way to assign global variables to specific addresses is (Listing 8.8).

### **Listing 8.8 Assigning global variables to specific addresses**

---

```
#pragma DATA_SEG [__SHORT_SEG] <segment_name>
```

---

setting the `PLACEMENT` section in the linker parameter file. An older method of accomplishing this is shown in Listing 8.9.



**Listing 8.9 Another means of assigning global variables to specific addresses**

---

```
<segment_name> INTO READ_ONLY <Address> ;
```

---

Listing 8.10 is a correct and incorrect example of using the global variable address modifier and Listing 8.11 is a possible PRM file that corresponds with example Listing.

**Listing 8.10 Using the global variable address modifier**

---

```
int glob @0x0500 = 10; // OK, global variable "glob" is
                        // at 0x0500, initialized with 10
void g() @0x40c0;      // error (the object is a function)

void f() {
    int i @0x40cc;     // error (the object is a local variable)
}
```

---

**Listing 8.11 Corresponding link parameter file settings (PRM file)**

---

```
/* the address 0x0500 of "glob" must not be in any address
   range of the SECTIONS entries */
SECTIONS
    MY_RAM      = READ_WRITE 0x0800 TO 0x1BFF;
    MY_ROM      = READ_ONLY  0x2000 TO 0xFEFF;
    MY_STACK    = READ_WRITE 0x1C00 TO 0x1FFF;
    MY_IO_SEG   = READ_WRITE 0x0400 TO 0x4ff;
END
PLACEMENT
    IO_SEG      INTO MY_IO_SEG;
    DEFAULT_ROM INTO MY_ROM;
    DEFAULT_RAM INTO MY_RAM;
    SSTACK     INTO MY_STACK;
END
```

---

## Variable Allocation using @ “SegmentName”

Sometimes it is useful to have the variable directly allocated in a named segment instead of using a #pragma. Listing 8.12 is an example of how to do this.

**Listing 8.12 Allocation of variables in named segments**

---

```
#pragma DATA_SEG __SHORT_SEG tiny
#pragma DATA_SEG not_tiny
```

---

## ANSI-C Frontend

### Implementation Features

---

```
#pragma DATA_SEG __SHORT_SEG tiny_b
#pragma DATA_SEG DEFAULT
int i@"tiny";
int j@"not_tiny";
int k@"tiny_b";
```

---

So with some pragmas in a common header file and with another definition for the macro, it is possible to allocate variables depending on a macro.

---

Declaration = <TypeSpec> <Declarator>[@<Section>"] [=<Initializer>];

---

Variables declared and defined with the @"section" syntax behave exactly like variables declared after their respective pragmas.

- <TypeSpec> is the type specifier, e.g., int or char
- <Declarator> is the identifier of your global object, e.g., i, glob
- <Section> is the section name. It should be defined in the link parameter file as well. For example, "MyDataSection".
- <Initializer> is the value to which the global variable is initialized.

The section name used has to be known at the declaration time by a previous section pragma (Listing 8.13).

#### Listing 8.13 Examples of section pragmas

---

```
#pragma DATA_SEG __SHORT_SEG MY_SHORT_DATA_SEG
#pragma DATA_SEG MY_DATA_SEG
#pragma CONST_SEC MY_CONST_SEC
#pragma DATA_SEG DEFAULT // not necessary, but is good practice
#pragma CONST_SEC DEFAULT // not necessary, but is good practice
int short_var @"MY_SHORT_DATA_SEG"; // OK, accesses are short
int ext_var @"MY_DATA_SEG" = 10; // OK, goes into
// MY_DATA_SECT
int def_var; / OK, goes into DEFAULT_RAM
const int cst_var @"MY_CONST_SEC" = 10; // OK, goes into MY_CONST_SECT
```

---

#### Listing 8.14 Corresponding Link Parameter File Settings (PRM file)

---

```
SECTIONS
MY_ZRAM = READ_WRITE 0x00F0 TO 0x00FF;
MY_RAM = READ_WRITE 0x0100 TO 0x01FF;
MY_ROM = READ_ONLY 0x2000 TO 0xFEFF;
```

---

```

    MY_STACK = READ_WRITE 0x0200 TO 0x03FF;
END
PLACEMENT
    MY_CONST_SEC, DEFAULT_ROM INTO MY_ROM;
    MY_SHORT_DATA_SEC INTO MY_ZRAM;
    MY_DATA_SEC, DEFAULT_RAM INTO MY_RAM;
    SSTACK INTO MY_STACK;
END

```

---

## Absolute Functions

Sometimes it is useful to call a absolute function (e.g., a special function in ROM). Listing 8.15 is a simple example of how this could be done using normal ANSI-C.

### Listing 8.15 Absolute function

---

```

#define erase ((void*)(void))(0xfc06)
void main(void) {
    erase(); /* call function at address 0xfc06 */
}

```

---

## Absolute Variables and Linking

Special attention is needed if absolute variables are involved in the linker's link process.

If an absolute object is not referenced by the application, the absolute variable is not linked in HIWARE format by default. Instead, it is always linked using the ELF/DWARF format. To force linking, switch off smart linking in the Linker, or using the ENTRIES command in the linker parameter file.

NOTE: Interrupt vector entries are always linked.

The example in Listing 8.16 shows how the linker handles different absolute variables.

### Listing 8.16 Linker Handling of Absolute Variables

---

```

    char i;          /* zero out          */
    char j = 1;     /* zero out, copy-down */
const char k = 2;  /* download   */
    char I@0x10;   /* no zero out! */
    char J@0x11 = 1; /* copy down   */
const char K@0x12 = 2; /* HIWARE: copy down / ELF: download! */
static char L@0x13; /* no zero out! */
static char M@0x14 = 3; /* copy down   */
static const char N@0x15 = 4; /* HIWARE: copy down, ELF: download */

```

---

## ANSI-C Frontend

### Implementation Features

---

```
void interrupt 2 MyISRfct(void) {} /* download, always linked! */
/* vector number two is downloaded with &MyISRfct */

void foo(char *p) {} /* download */

void main(void) { /* download */
    foo(&i); foo(&j); foo(&k);
    foo(&I); foo(&J); foo(&K);
    foo(&L); foo(&M); foo(&N);
}
```

---

Zero out means that the default startup code initializes the variables during startup. Copy down means that the variable is initialized during the default startup. To download means that the memory is initialized while downloading the application.

## The `__far` Keyword

The keyword `far` is a synonym for `__far`, which is not allowed when the `-Ansi: Strict ANSI` compiler option is present.

---

**NOTE** Not all Backends may support this keyword. See the Non-ANSI Keywords section in the HC08 Backend.

---

A `__far` pointer allows access to the whole memory range supported by the processor, not just to the default data page. You can use it to access memory mapped I/O registers that are not on the data page. You can also use it to allocate constant strings in a ROM not on the data page.

The '`__far`' keyword defines the calling convention for a function. Some backends support special calling conventions which also set a page register when a function is called. This enables you to use more code than the address space can usually accommodate. The special allocation of such functions is not done automatically.

## Using the `__far` Keyword for Pointers

The keyword `__far` is a type qualifier like `const` and is valid only in the context of pointer types and functions. The `__far` keyword (for pointers) always affects the last '\*' to its left in a type definition. The declaration of a `__far` pointer to a `__far` pointer to a character is:

```
char *__far *__far p;
```

The following is a declaration of a normal (short) pointer to a `__far` pointer to a character:

```
char *__far * p;
```

---

**NOTE** To declare a `__far` pointer, place the `__far` keyword *after* the asterisk:

```
char *__far p;
```

not

```
char __far *p;
```

The second choice will not work.

---

## `__far` and Arrays

The `__far` keyword does not appear in the context of the `'*` type constructor in the declaration of an array parameter, as shown:

```
void my_func (char a[37]);
```

Such a declaration specifies a pointer argument. This is equal to:

```
void my_func (char *a);
```

There are two possible uses when declaring such an argument to a `__far` pointer:

```
void my_func (char a[37] __far);
```

or alternately

```
void my_func (char *__far a);
```

In the context of the `'[]` type constructor in a direct parameter declaration, the `__far` keyword always affects the first dimension of the array to its left. In the following declaration, parameter `a` has type “`__far` pointer to array of 5 `__far` pointers to `char`”:

```
void my_func (char *__far a[][5] __far);
```

## `__far` and typedef Names

If the array type has been defined as a `typedef` name, as in:

```
typedef int ARRAY[10];
```

then a `__far` parameter declaration is:

```
void my_func (ARRAY __far a);
```

The parameter is a `__far` pointer to the first element of the array. This is equal to:

```
void my_func (int *__far a);
```

It is also equal to the following direct declaration:

```
void my_func (int a[10] __far);
```

It is *not* the same as specifying a `__far` pointer to the array:

```
void my_func (ARRAY *__far a);
```

because `a` has type “`__far` pointer to `ARRAY`” instead of “`__far` pointer to `int`”.

## **\_\_far and Global Variables**

The `__far` keyword can also be used for global variables:

```
int __far i;           // OK for global variables
int __far *i;         // OK for global variables
int __far *__far i;   // OK for global variables
```

This forces the Compiler to perform the same addressing mode for this variable as if it has been declared in a `__FAR_SEG` segment. Note that for the above variable declarations or definitions, the variables are in the `DEFAULT_DATA` segment if no other data segment is active. Be careful if you mix ‘`__far`’ declarations or definitions within a non-`__FAR_SEG` data segment. Assuming that `__FAR_SEG` segments have ‘extended’ addressing mode and normal segments have ‘direct’ addressing mode, the following two examples (Listing 8.17 and Listing 8.18) clarify this behavior:

### **Listing 8.17 OK, consistent declarations**

---

```
#pragma DATA_SEG MyDirectSeg           // use direct addressing mode
int i;                                   // direct, segment MyDirectSeg
int j;                                   // direct, segment MyDirectSeg
#pragma DATA_SEG __FAR_SEG MyFarSeg /* use extended addressing mode */
int k;                                   // extended, segment MyFarSeg
int l;                                   // extended, segment MyFarSeg
int __far m; // extended, segment MyFarSeg
```

---

**Listing 8.18 Mixing extended addressing and direct addressing modes**

---

```
// caution: not consistent!!!!
#pragma DATA_SEG MyDirectSeg /* use direct-addressing mode */
int i;          // direct, segment MyDirectSeg
int j;          // direct, segment MyDirectSeg
int __far k;    // extended, segment MyDirectSeg
int __far l;    // extended, segment MyDirectSeg
int __far m;    // extended, segment MyDirectSeg
```

---

**NOTE** The `__far` keyword global variables only affect the access to the variable (addressing mode) and NOT the allocation.

---

## **`__far` and C++ Classes**

If a member function gets the modifier `__far`, the “this” pointer is a `__far` pointer. This is useful, if for instance, if the owner class of the function is not allocated on the default data page. See Listing 8.19.

**Listing 8.19 `__far` member functions**

---

```
class A {
public:
    void f_far(void) __far {
        /* __far version of member function A::f() */
    }
    void f(void) {
        /* normal version of member function A::f() */
    }
};
#pragma DATA_SEG MyDirectSeg // use direct addressing mode
A a_normal; // normal instance
#pragma DATA_SEG __FAR_SEG MyFarSeg // use extended addressing mode
A __far a_far; // __far instance
void main(void)
    a_normal.f(); // call normal version of A::f() for normal instance
    a_far.f_far(); // call __far version of A::f() for __far instance
}
```

---

## \_\_far and C++ References

The `__far` modifier is applied to references. This is useful if it is a reference to an object outside of the default data page. See Listing 8.20.

### Listing 8.20 `__far` modifier applied to references

---

```
int j; // object j allocated outside the default data page
      // (must be specified in the link parameter file)
void f(void) {
    int &__far i = j;
};
```

---

## Using the `__far` Keyword for Functions

A special calling convention is specified for the `__far` keyword. The `__far` keyword is specified in front of the function identifier:

```
void __far f(void);
```

If the function returns a pointer, the `__far` keyword must be written in front of the first asterisk (“\*”).

```
int __far *f(void);
```

It must, however, be after the `int` and not before it.

For function pointers, many backends assume that the `__far` function pointer is pointing to functions with the `__far` calling convention, even if the calling convention was not specified. Moreover, most backends do not support different function pointer sizes in one compilation unit. The function pointer size is then dependent only upon the memory model. See the backend chapter for details.

**Table 8.1** Interpretation of the `__far` Keyword

Declaration	Allowed	Type Description
<code>int __far f();</code>	OK	<code>__far</code> function returning an int
<code>__far int f();</code>	error	
<code>__far f();</code>	OK	<code>__far</code> function returning an int
<code>int __far *f();</code>	OK	<code>__far</code> function returning a pointer to int
<code>int * __far f();</code>	OK	function returning a <code>__far</code> pointer to int



**Table 8.1 Interpretation of the `__far` Keyword (continued)**

Declaration	Allowed	Type Description
<code>__far int * f();</code>	error	
<code>int __far * __far f();</code>	OK	<code>__far</code> function returning a <code>__far</code> pointer to int
<code>int __far i;</code>	OK	global <code>__far</code> object
<code>int __far *i;</code>	OK	pointer to a <code>__far</code> object
<code>int * __far i;</code>	OK	<code>__far</code> pointer to int
<code>int __far * __far i;</code>	OK	<code>__far</code> pointer to a <code>__far</code> object
<code>__far int *i;</code>	OK	pointer to a <code>__far</code> integer
<code>int * __far (* __far f)(void)</code>	OK	<code>__far</code> pointer to function returning a <code>__far</code> pointer to int
<code>void * __far (* f)(void)</code>	OK	pointer to function returning a <code>__far</code> pointer to void
<code>void __far * (* f)(void)</code>	OK	pointer to <code>__far</code> function returning a pointer to void

## `__near` Keyword

**NOTE** See the *Non-ANSI Keywords* section in the HC(S)08 Backend. The `near` keyword is a synonym for `__near`. The `near` keyword is only allowed when the `-Ansi: Strict ANSI` compiler option is present.

The `__near` keyword can be used instead of the `__far` keyword. It is used in situations where non-qualified pointers are `__far` and an explicit `__near` access should be specified or where the `__near` calling convention must be explicitly specified.

The `__near` keyword uses two semantic variations. Either it specifies a small size of a function or data pointers or it specifies the `__near` calling convention.

**Table 8.2 Interpretation of the `__near` Keyword**

Declaration	Allowed	Type Description
<code>int __near f();</code>	OK	<code>__near</code> function returning an int
<code>int __near __far f();</code>	error	

**Table 8.2 Interpretation of the `__near` Keyword (continued)**

Declaration	Allowed	Type Description
<code>__near f();</code>	OK	<code>__near</code> function returning an <code>int</code>
<code>int __near * __far f();</code>	OK	<code>__near</code> function returning a <code>__far</code> pointer to <code>int</code>
<code>int __far *i;</code>	error	
<code>int * __near i;</code>	OK	<code>__far</code> pointer to <code>int</code>
<code>int * __far* __near i;</code>	OK	<code>__near</code> pointer to <code>__far</code> pointer to <code>int</code>
<code>int * __far (* __near f)(void)</code>	OK	<code>__near</code> pointer to function returning a <code>__far</code> pointer to <code>int</code>
<code>void * __near (* f)(void)</code>	OK	pointer to function returning a <code>__near</code> pointer to <code>void</code>
<code>void __far * __near (* __near f)(void)</code>	OK	<code>__near</code> pointer to <code>__far</code> function returning a <code>__far</code> pointer to <code>void</code>

## Compatibility

`__far` pointers and normal pointers are compatible. If necessary, the normal pointer is extended to a `__far` pointer (subtraction of two pointers or assignment to a `__far` pointer). In the other case, the `__far` pointer is clipped to a normal pointer (i.e., the page part is discarded).

## `__alignof__` Keyword

Some processors align objects according to their type. The unary operator, `__alignof__`, determines the alignment of a specific type. By providing any type, this operator returns its alignment. This operator behaves in the same way as "`sizeof(type_name)`" operator. See the target backend section to check which alignment corresponds to which fundamental data type (if any is required) or to which aggregate type (structure, array).

This macro may be useful for the `va_arg` macro in `stdarg.h`, e.g., to differentiate the alignment of a structure containing four objects of four bytes from that of a structure containing two objects of eight bytes. In both cases, the size of the structure is 16 bytes, but the alignment may differ, as shown in Listing 8.21:

**Listing 8.21 va\_arg macro**

---

```
#define va_arg(ap,type) \
  (((__alignof__(type)>=8) ? \
   ((ap) = (char *)(((int)(ap) \
    + __alignof__(type) - 1) & (~(__alignof__(type) - 1)))) \
   : 0), \
  ((ap) += __va_rounded_size(type)), \
  ((type *) (ap))[-1]))
```

---

## **\_\_va\_sizeof\_\_ Keyword**

According to the ANSI-C specification, you must promote character arguments in open parameter lists to int. The use of “char” in the va\_arg macro to access this parameter may not work as per the ANSI-C specification (Listing 8.22).

**Listing 8.22 Inappropriate use of char with the va\_arg macro**

---

```
int f(int n, ...) {
    int res;
    va_list l= va_start(n, int);
    res= va_arg(l, char); /* should be va_arg(l, int) */
    va_end(l);
    return res;
}

void main(void) {
    char c=2;
    int res=f(1,c);
}
```

---

With the \_\_va\_sizeof\_\_ operator, the va\_arg macro is written the way that f returns 2.

A safe implementation of the f function is to use “va\_arg(l, int)” instead of “va\_arg(l, char)”.

The \_\_va\_sizeof\_\_ unary operator, which is used exactly as the sizeof keyword, returns the size of its argument after promotion as in an open parameter list (Listing 8.23).

**Listing 8.23 \_\_va\_sizeof\_\_ examples**

---

```
__va_sizeof__(char) == sizeof (int)
__va_sizeof__(float) == sizeof (double)
```

---

## ANSI-C Frontend

### Implementation Features

---

```
struct A { char a; };
__va_sizeof__(struct A) >= 1 (1 if the target needs no padding bytes)
```

---

**NOTE** It is not possible in ANSI-C to distinguish a 1-byte structure without alignment or padding from a character variable in a `va_arg` macro. They need a different space on the open parameter calls stack for some processors.

---

## interrupt Keyword

The `__interrupt` keyword is a synonym for `interrupt`, which is allowed when the `-Ansi:Strict` ANSI compiler option is present.

**NOTE** See the *Non-ANSI Keywords* section in the HS(S)08 Backend. One of two ways can be used to specify a function as an interrupt routine:

---

1. Use `#pragma TRAP_PROC`: Mark function as interrupt Function and adapt the Linker parameter file.
2. Use the nonstandard interrupt keyword.

Use the nonstandard interrupt keyword like any other type qualifier (Listing 8.24). It specifies a function to be an interrupt routine. It is followed by a number specifying the entry in the interrupt vector that should contain the address of the interrupt routine. If it is not followed by any number, the interrupt keyword has the same effect as the `TRAP_PROC` pragma. It specifies a function to be an interrupt routine. However, the number of the interrupt vector must be associated with the name of the interrupt function by using the Linker's `VECTOR` directive in the Linker parameter file.

### Listing 8.24 Examples of the interrupt keyword

---

```
interrupt void f(); // OK
// same as #pragma TRAP_PROC,
// please set the entry number in the prm-file
interrupt 2 int g();
// The 2nd entry (number 2) gets the address of func g().
interrupt 3 int g(); // OK
// third entry in vector points to g()
interrupt int l; // error: not a function
```

---

## \_\_asm Keyword

The Compiler supports target processor instructions inside of C functions.

---

The `asm` keyword is a synonym for `__asm`, which is allowed when the `-Ansi: Strict` ANSI compiler option is not present (Listing 8.25).

See the inline assembler section in the backend chapter for details.

**Listing 8.25 Examples of the `__asm` keyword**

---

```
__asm {
  nop
  nop ; comment
}
asm ("nop; nop");
__asm("nop\n nop");
__asm "nop";
__asm nop;
#asm
  nop
  nop
#endasm
```

---

## Intrinsic Functions

ANSI-C does not provide a mechanism to efficiently read a processor flag.

### Read Processor Flags

To avoid using HLI for this purpose, the Compiler offers a set of intrinsic functions. The code of these functions is inlined. The processor flags listed in Table 8.3 are read by the associated intrinsic function.

**Table 8.3 Read Processor Flags**

Flag	Flag Abbreviation	Intrinsic Function Name
Carry	C	<code>__isflag_carry()</code>
Half carry	H	<code>__isflag_half_carry()</code>
Overflow	V	<code>__isflag_overflow()</code>
Interrupt pin high	I	<code>__isflag_int()</code>
Interrupt enable	M	<code>__isflag_int_enabled()</code>

Example:

```
if(__isflag_carry()) goto label
```

translates to a conditional branch to 'label', i.e., branches if the carry flag are set (for HC08, the resulting code is `BCS label`).

## Implementation-Defined Behavior

The ANSI standard contains a couple of places where the behavior of a particular Compiler is left undefined. It is possible for different Compilers to implement certain features in different ways, even if they all comply with the ANSI-C standard. Subsequently, the following discuss those points and the behavior implemented by the Compiler.

### Right Shifts

The result of `E1 >> E2` is implementation-defined for a right shift of an object with a signed type having a negative value if `E1` has a signed type and a negative value.

In this implementation, an arithmetic right shift is performed.

### Initialization of Aggregates with non Constants

The initialization of aggregates with non-constants is not allowed in the ANSI-C specification. The Compiler allows it if the `-Ansi: Strict ANSI compiler` option is not set (see Listing 8.26).

#### Listing 8.26 Initialization using a non constant

---

```
void main() {  
    struct A {  
        struct A *n;  
    } v={&v}; /* the address of v is not constant */  
}
```

---

### Sign of char

The ANSI-C standard leaves it open, whether the data type `char` is signed or unsigned. Check the Backend chapter for data about default settings.

### Division and Modulus

The results of the `" / "` and `" % "` operators are also not properly defined for signed arithmetic operations unless both operands are positive.

**NOTE** The way a Compiler implements " / " and "% " for negative operands is determined by the hardware implementation of the target's division instructions.

---

## Translation Limitations

This section describes the internal limitations of the Compiler. Some limitations are stack limitations depending on the operating system used. For example, in some operating systems, limits depend on whether the compiler is a 32-bit compiler running on a 32-bit platform (e.g., Windows NT), or if it is a 16-bit Compiler running on a 16-bit platform (e.g., Windows for Workgroups).

The ANSI-C column in Table 8.4 below shows the recommended limitations of ANSI-C (5.2.4.1 in ISO/IEC 9899:1990 (E)) standard. These quantities are only guidelines and do not determine compliance. The 'Implementation' column shows the actual implementation value and the possible message number. '-' means that there is no information available for this topic and 'n/a' denotes that this topic is not available.

**Table 8.4 Translation Limitations (ANSI)**

Limitation	Implementation	ANSI-C
Nesting levels of compound statements, iteration control structures, and selection control structures	256 (C1808)	15
Nesting levels of conditional inclusion	-	8
Pointer, array, and function decorators (in any combination) modifying an arithmetic, structure, union, or incomplete type in a declaration	-	12
Nesting levels of parenthesized expressions within a full expression	32 (C4006)	32
Number of initial characters in an internal identifier or macro name	32,767	31
Number of initial characters in an external identifier	32,767	6
External identifiers in one translation unit	-	511
Identifiers with block scope declared in one block	-	127

**Table 8.4 Translation Limitations (ANSI) (continued)**

<b>Limitation</b>	<b>Implementation</b>	<b>ANSI-C</b>
Macro identifiers simultaneously defined in one translation unit	655,360,000 (C4403)	1024
Parameters in one function definition	-	31
Arguments in one function call	-	31
Parameters in one macro definition	1024 (C4428)	31
Arguments in one macro invocation	2048 (C4411)	31
Characters in one logical source line	2 <sup>31</sup>	509
Characters in a character string literal or wide string literal (after concatenation)	8196 (C3301, C4408, C4421)	509
Size of an object	32,767	32,767
Nesting levels for #include files	512 (C3000)	8
Case labels for a switch statement (excluding those for any nested switch statements)	1000	257
Data members in a single class, structure, or union	-	127
Enumeration constants in a single enumeration	-	127
Levels of nested class, structure, or union definitions in a single struct declaration list	32	15
Functions registered by atexit()	-	n/a
Direct and indirect base classes	-	n/a
Direct base classes for a single class	-	n/a
Members declared in a single class	-	n/a
Final overriding virtual functions in a class, accessible or not	-	n/a
Direct and indirect virtual bases of a class	-	n/a
Static members of a class	-	n/a



**Table 8.4 Translation Limitations (ANSI) (continued)**

Limitation	Implementation	ANSI-C
Friend declarations in a class	-	n/a
Access control declarations in a class	-	n/a
Member initializers in a constructor definition	-	n/a
Scope qualifications of one identifier	-	n/a
Nested external specifications	-	n/a
Template arguments in a template declaration	-	n/a
Recursively nested template instantiations	-	n/a
Handlers per try block	-	n/a
Throw specifications on a single function declaration	-	n/a

The table below shows other limitations which are not mentioned in an ANSI standard:

**Table 8.5 Translation Limitations (non-ANSI)**

Limitation	Description
Type Declarations	Derived types must not contain more than 100 components.
Labels	There may be at most 16 other labels within one procedure.
Macro Expansion	Expansion of recursive macros is limited to 70 (16-bit OS) or 2048 (32-bit OS) recursive expansions (C4412).
Include Files	The total number of include files is limited to 8196 for a single compilation unit.
Numbers	Maximum of 655,360,000 different numbers for a single compilation unit (C2700, C3302).
Goto	M68k only: Maximum of 512 Gotos for a single function (C15300).
Parsing Recursion	Maximum of 1024 parsing recursions (C2803).
Lexical Tokens	Limited by memory only (C3200).

**Table 8.5 Translation Limitations (non-ANSI) (*continued*)**

<b>Limitation</b>	<b>Description</b>
Internal IDs	Maximum of 16,777,216 internal IDs for a single compilation unit (C3304). Internal IDs are used for additional local or global variables created by the Compiler (e.g., by using CSE).
Code Size	Code size is limited to 32KB for each single function.
filenames	Maximum length for filenames (including path) are 128 characters for 16-bit applications or 256 for Win32 applications. UNIX versions support filenames without path of 64 characters in length and 256 in the path. Paths may be 96 characters on 16-bit PC versions, 192 on UNIX versions or 256 on 32-bit PC versions.

## **ANSI-C Standard**

This section provides a short overview about the implementation (see also ANSI Standard 6.2) of the ANSI-C conversion rules.

### **Integral Promotions**

You may use a `char`, a `short int`, or an `int` bitfield, or their signed or unsigned varieties, or an `enum` type, in an expression wherever an `int` or `unsigned int` is used. If an `int` represents all values of the original type, the value is converted to an `int`; otherwise, it is converted to an `unsigned int`. Integral promotions preserve the value including its sign.

### **Signed and Unsigned Integers**

Promoting a signed integer type to another signed integer type of greater size requires "sign extension". In two's-complement representation, the bit pattern is unchanged, except for filling the high order bits with copies of the sign bit.

When converting a signed integer type to an unsigned integer type, if the destination has equal or greater size, the first signed extension of the signed integer type is performed. If the destination has a smaller size, the result is the remainder on division by a number, one greater than the largest unsigned number, that is represented in the type with the smaller size.

## Arithmetic Conversions

The operands of binary operators do implicit conversions:

- If either operand has type `long double`, the other operand is converted to `long double`.
- If either operand has type `double`, the other operand is converted to `double`.
- If either operand has type `float`, the other operand is converted to `float`.
- The integral promotions are performed on both operands.

Then the following rules are applied:

- If either operand has type `unsigned long int`, the other operand is converted to `unsigned long int`.
- If one operand has type `long int` and the other has type `unsigned int`, if a `long int` can represent all values of an `unsigned int`, the operand of type `unsigned int` is converted to `long int`; if a `long int` cannot represent all the values of an `unsigned int`, both operands are converted to `unsigned long int`.
- If either operand has type `long int`, the other operand is converted to `long int`.
- If either operand has type `unsigned int`, the other operand is converted to `unsigned int`.
- Both operands have type `int`.

## Order of Operand Evaluation

The priority order of operators and their associativity is listed in Listing 8.27.

**Listing 8.27 Operator precedence**

Operators	Associativity
<code>() [] -&gt; .</code>	left to right
<code>! ~ ++ -- + - * &amp; (type) sizeof</code>	right to left
<code>&amp; / %</code>	left to right
<code>+ -</code>	left to right
<code>&lt;&lt; &gt;&gt;</code>	left to right
<code>&lt; &lt;= &gt; &gt;=</code>	left to right
<code>== !=</code>	left to right
<code>&amp;</code>	left to right
<code>^</code>	left to right
<code> </code>	left to right
<code>&amp;&amp;</code>	left to right
<code>  </code>	left to right
<code>? :</code>	right to left
<code>= += -= *= /= %= &amp;= ^=  = &lt;&lt;= &gt;&gt;=</code>	right to left

## ANSI-C Frontend

### Floating-Type Formats

---

, left to right

---

Unary +, - and \* have higher precedence than the binary forms.

Listing 8.28 has some examples of operator precedence

#### Listing 8.28 Examples of operator precedence

---

```
if (a&3 == 2)
'==' has higher precedence than '&', thus it is evaluated as:
if (a & (3==2))
```

which is the same as:  
if (a&0)

---

---

**TIP** Use brackets if you are not sure about associativity!

---

## Rules for Standard-Type Sizes

In ANSI-C, enumerations have the type of 'int'. In this implementation they have to be smaller than or equal to 'int'. Listing 8.29 lists the size rules for integer types.

#### Listing 8.29 Size relationships among the integer types

---

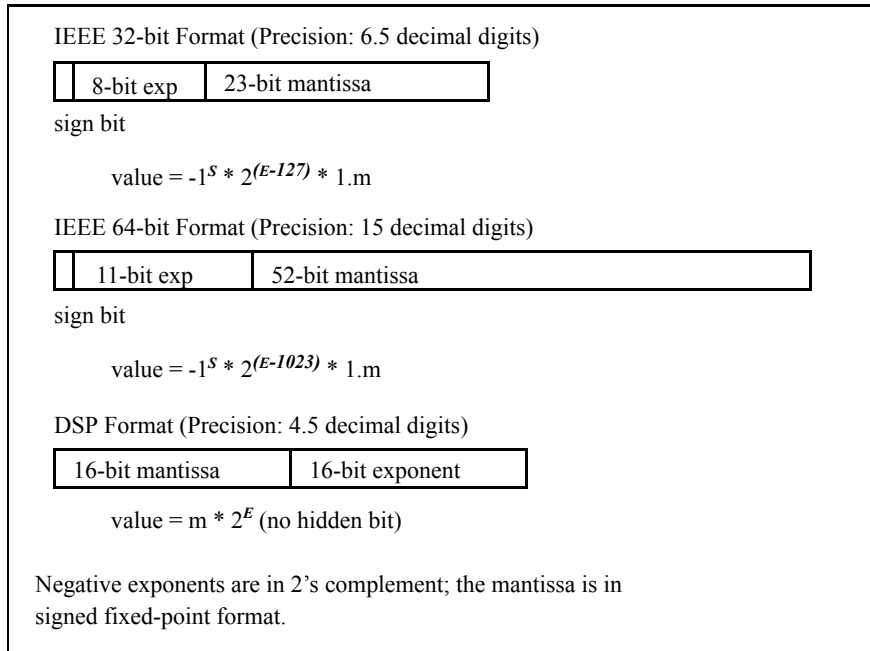
```
sizeof(char)   <= sizeof(short)
sizeof(short)  <= sizeof(int)
sizeof(int)    <= sizeof(long)
sizeof(long)   <= sizeof(long long)
sizeof(float)  <= sizeof(double)
sizeof(double) <= sizeof(long double)
```

---

## Floating-Type Formats

The Compiler supports two IEEE floating point formats: IEEE32 and IEEE64. There may also be a DSP format supported by the processor. Figure 8.1 shows these three formats.

Figure 8.1 Floating-point formats



Floats are implemented as IEEE32 and doubles as IEEE64. This may vary for a specific Backend, or possibly, both formats may not be supported. Please check the Backend chapter for details, default settings and supported formats.

## Floating-Point Representation of 500.0 for IEEE

First, convert 500.0 from the decimal representation to a representation with base 2:

$$\text{value} = (-1)^s * m * 2^{\text{exp}}$$

where: s, sign is 0 or 1,  
 $2 > m \geq 1$  for IEEE,  
 and exp is a integral number.

For 500, this gives:

$$\begin{aligned} \text{sign}(500.0) &= 1, \\ m, \text{ mant}(500.0, \text{IEEE}) &= 1.953125, \text{ and} \end{aligned}$$

## ANSI-C Frontend

### Floating-Type Formats

---

`exp (500.0, IEEE) = 8`

---

**NOTE** The number 0 (zero) cannot be represented this way. So for 0, IEEE defines a special bit pattern consisting of 0 bits only.

---

Next, convert the mantissa into its binary representation.

```
mant (500.0, IEEE) = 1.953125
= 1*2^(0) + 1*2^(-1) + 1*2^(-2) + 1*2^(-3) + 1*2^(-4)
  + 0*2^(-5) + 1*2^(-6) + 0*...
= 1.111101000... (binary)
```

Because this number is converted to be larger or equal to 1 and smaller than 2, there is always a 1 in front of the decimal point. For the remaining steps, this constant (1) is left out in order to save space.

```
mant (500.0, IEEE, cut) = .111101000...
```

The exponent must also be converted to binary format:

```
exp (500.0, IEEE) = 8 == 08 (hex) == 1000 (binary)
```

For the IEEE formats, the sign is encoded as a separate bit (sign magnitude representation)

## Representation of 500.0 in IEEE32 Format

The exponent in IEEE32 has a fixed offset of 127 in order to always have positive values:

```
exp (500.0, IEEE32) = 8 + 127 == 87 (hex) == 1000111 (bin)
```

The fields must be put together as shown Listing 8.30:

### Listing 8.30 Representation of decimal 500.0 in IEEE32

---

```
= 0 (sign) 10000111 (exponent)
  111101000000000000000000 (mantissa)      (IEEE32 as bin)
= 0100 0011 1111 1010 0000 0000 0000 0000 (IEEE32 as bin)
= 43 fa 00 00 (IEEE32 as hex)
```

---

The IEEE32 representation of decimal -500 is shown in Listing 8.31.



documentation for details.

Except for the 0 (zero) and -0 (minus zero) special formats, not all special formats may be supported for specific backends.

---

## Representation of 500.0 in DSP Format

Convert 500.0 from the decimal representation to a representation with base 2. In contrast to IEEE, DSP normalizes the mantissa between 0 and 1 and not between 1 and 2. This makes it possible to also represent 0, which must have a special pattern in IEEE. In addition, the exponent is different from IEEE.

$$\text{value} = (-1)^s * m * 2^e$$

where:

- $s$ , sign, is 1 or -1,
- $1 > m \geq 0$ , and
- $e$  is an integral number.

For 500 this gives:

- $\text{sign}(500.0) = 1$
- $\text{mant}(500.0, \text{DSP}) = 0.9765625$
- $\text{exp}(500.0, \text{DSP}) = 9$

Next convert the mantissa into its binary representation (Listing 8.34).

### Listing 8.34 Representation of 500 in DSP format

---

```
mant (500.0, DSP) = 0.9765625 (dec)
= 0*2^(0) + 1*2^(-1) + 1*2^(-2) + 1*2^(-3) + 1*2^(-4)
  + 1*2^(-5) + 0*2^(-6) + 1*2^(-7) + 0*...
= 0.1111101000... (bin)
```

---

Because this number is computed to always be larger or equal to 0 and smaller than 1, there is always a 0 in front of the decimal point. For the remaining steps this constant is omitted in order to save space. There is always a 1 after the decimal point, except for 0 and intermediate results. This bit is encoded so the DSP loses one additional bit of precision compared with IEEE.

```
mant (500.0, DSP, cut) = .1111101000...
```

The exponent must also be converted to binary format:

```
exp (500.0, DSP) = 9 == 09 (hex) == 1001 (bin)
```

---



Negative exponents are encoded by the 2's representation of the positive value.

The sign is encoded into the mantissa by taking the 2's complement for negative numbers and adding a 1 bit in the front. For DSP and positive numbers, a 0 bit is added at the front.

```
mant(500.0, DSP) = 0111110100000000 (bin)
```

The twos complement is taken for negative numbers:

```
mant(-500.0, DSP) = 1000001100000000 (bin)
```

Finally the mantissa and the exponent must be joined according to Figure 8.1:

---

**Listing 8.35 Representation of decimal 500.0 in DSP**

---

```
= 7D 00 (mantissa) 00 09 (exponent) (DSP as hex)
= 7D 00 00 09 (DSP as hex)
= 0111 1101 0000 0000 0000 0000 0000 1001 (DSP as binary)
```

---

The DSP representation of decimal -500 is shown in Listing 8.36.

---

**Listing 8.36 Representation of decimal -500.0 in DSP**

---

```
= 83 00 (mantissa) 00 09 (exponent) (DSP as hex)
= 83 00 00 09 (DSP as hex)
= 1000 0011 0000 0000 0000 0000 0000 1001 (DSP as binary)
```

---

---

**NOTE** The order of the byte representation of a floating-point value depends on the byte ordering of the backend. The first byte in the previous diagrams must be considered as the most significant byte.

---

## Volatile Objects and Absolute Variables

The Compiler does not do register- and constant tracing on volatile or absolute global objects. Accesses to volatile or absolute global objects are not eliminated. See Listing 8.37 for one reason to use a volatile declaration.

---

**Listing 8.37 Using volatile to avoid an adverse side effect**

---

```
volatile int x;
void main(void) {
    x = 0;
```

---

## ANSI-C Frontend

### Bitfields

---

```
...
if (x == 0) { // without volatile attribute, the
              // comparison may be optimized away!
    Error(); // Error() is called without compare!
}
}
```

---

## Bitfields

There is no standard way to allocate bitfields. Bitfield allocation varies from compiler to compiler, even for the same target. Using bitfields for access to I/O registers is non-portable and inefficient for the masking involved in unpacking individual fields. It is recommended that you use regular bit-and (&) or bit-or (|) operations for I/O port access.

The maximum width of bitfields is backend-dependent (see the HC(S)08 Backend chapter for details), in that plain `int` bitfields are signed. As stated in Kernighan and Ritchie's *"The C Programming Language"*, 2<sup>ND</sup> ed., the use of bitfields is equivalent to using bit masks to which the operators `&`, `|`, `~`, `|=`, or `&=` are applied. In fact, the Compiler translates bitfield operations to bit mask operations.

## Signed Bitfields

A common mistake is to use signed bitfields, but testing them as if they were unsigned. Signed bitfields have a value -1 or 0. Consider the following example: (Listing 8.38).

### Listing 8.38 Testing a signed bitfield as being unsigned

---

```
typedef struct _B {
    signed int b0: 1;} B;
    B b;
if (b.b0 == 1) ...
```

---

The Compiler issues a warning and replaces the 1 with -1 because the condition `(b.b0 == 1)` does not make sense, i.e., it is always false. The test `(b.b0 == -1)` is performed as expected. This substitution is not ANSI compatible and will not be performed when the `-Ansi: Strict ANSI` compiler option is active.

The correct way to specify this is with an unsigned bitfield. Unsigned bitfields have the values 0 or 1 (Listing 8.39).

---

**Listing 8.39 Using unsigned bitfields**

---

```
typedef struct _B {
    unsigned b0: 1;
} B;
B b;
if (b.b0 == 1) ...
```

---

Because b0 is an unsigned bitfield having a value 0 or 1, the test (b.b0 == 1) is correct.

## Recommendations

In order to save memory, it is recommended to implement globally accessible boolean flags as unsigned bitfields of width 1. However, it is not recommended using bitfields for other purposes because using bitfields to describe a bit pattern in memory is not portable between compilers, even on the same target, as different compilers may allocate bitfields differently.

For information about how the Compiler allocates bitfields, see the Data Types section in the HC(S)08 Backend.

# Segmentation

The Linker supports the concept of segments in that the memory space may be partitioned into several segments. The Compiler allows attributing a certain segment name to certain global variables or functions which then will be allocated into that segment by the Linker. Where that segment actually lies is determined by an entry in the Linker parameter file.

**Listing 8.40 The syntax of the segment specification pragma**

---

```
SegDef= "#pragma" SegmentType ({SegmentMod} SegmentName |
                                DEFAULT)
SegmentType= CODE_SEG|CODE_SECTION|
              DATA_SEG|DATA_SECTION|
              CONST_SEG|CONST_SECTION|
              STRING_SEG|STRING_SECTION

SegmentMod= __DIRECT_SEG|__NEAR_SEG|__CODE_SEG
            |__FAR_SEG|__BIT_SEG|__Y_BASED_SEG
            |__Z_BASED_SEG|__DPAGE_SEG|__PPAGE_SEG
            |__EPAGE_SEG|__RPAGE_SEG|__GPAGE_SEG"
            |__PIC_SEG|CompatSegmentMod
```

---

## ANSI-C Frontend

### Segmentation

---

CompatSegmentMod=DIRECT | NEAR | CODE | FAR | BIT | Y\_BASED | Z\_BASED |  
DPAGE | PPAGE | EPAGE | RPAGE | GPAGE | PIC

---

Because there are two basic types of segments, code and data segments, there are also two pragmas to specify segments:

- `#pragma CODE_SEG <segment_name>`
- `#pragma DATA_SEG <segment_name>`

In addition there are pragmas for constant data and for strings:

- `#pragma CONST_SEG <segment_name>`
- `#pragma STRING_SEG <segment_name>`

All four pragmas are valid until the next pragma of the same kind is encountered.

In the HIWARE object file format, constants are put into `DATA_SEG` if no `CONST_SEG` was specified. In the ELF Object file format, constants are always put into a constant segment.

Strings are put into the `STRINGS` segment until a `STRING_SEG` pragma is specified. After this pragma, all strings are allocated into this constant segment. The linker then treats this segment like any other constant segment.

If no segment is specified, the Compiler assumes two default segments named `DEFAULT_ROM` (the default code segment) and `DEFAULT_RAM` (the default data segment). Use the segment name `DEFAULT` to explicitly make these default segments the current segments:

```
#pragma CODE_SEG DEFAULT
#pragma DATA_SEG DEFAULT
#pragma CONST_SEG DEFAULT
#pragma STRING_SEG DEFAULT
```

Segments may also be declared as `__SHORT_SEG` by inserting the keyword `__SHORT_SEG` just before the segment name (with the exception of the predefined segment `DEFAULT` – this segment cannot be qualified with `__SHORT_SEG`). This makes the Compiler use short (i.e., 8 bits or 16 bits, depending on the Backend) absolute addresses to access global objects, or to call functions. It is the programmer's responsibility to allocate `__SHORT_SEG` segments in the proper memory area.

---

**NOTE** The default code and data segments may not be declared as `__SHORT_SEG`.

---

The meaning of the other segment modifiers, such as `__NEAR_SEG` and `__FAR_SEG`, are backend-specific. Modifiers that are not supported by the backend are ignored. Please refer to the backend chapter for data about which modifiers are supported.

The segment pragmas also have an effect on static local variables. Static local variables are local variables with the ‘static’ flag set. They are in fact normal global variables but with scope only to the function in which they are defined:

---

```
#pragma DATA_SEG MySeg

static char foo(void) {
    static char i = 0; /* place this variable into MySeg */
    return i++;
}

#pragma DATA_SEG DEFAULT
```

---

---

**NOTE** Using the ELF/DWARF object file format (-F1 or -F2 compiler option), all constants are placed into the section `.rodata` by default unless a `#pragma CONST_SEG` is used.

---

---

**NOTE** There are aliases to satisfy the ELF naming convention for all segment names. Use `CODE_SECTION` instead of `CODE_SEG`. Use `DATA_SECTION` instead of `DATA_SEG`. Use `CONST_SECTION` instead of `CONST_SEG`. Use `STRING_SECTION` instead of `STRING_SEG`. These aliases behave exactly as do the `XXX_SEG` name versions.

---

#### **Listing 8.41 Example of Segmentation without the -Cc Compiler Option**

---

```
static int a; /* Placed into Segment: */
static const int c0 = 10; /* DEFAULT_RAM(-1) */
/* DEFAULT_RAM(-1) */

#pragma DATA_SEG MyVarSeg
static int b; /* MyVarSeg(0) */
static const int c1 = 11; /* MyVarSeg(0) */

#pragma DATA_SEG DEFAULT
static int c; /* DEFAULT_RAM(-1) */
static const int c2 = 12; /* DEFAULT_RAM(-1) */

#pragma DATA_SEG MyVarSeg
#pragma CONST_SEG MyConstSeg
static int d; /* MyVarSeg(0) */
static const int c3 = 13; /* MyConstSeg(1) */

#pragma DATA_SEG DEFAULT
```

---

## ANSI-C Frontend

### Optimizations

---

```
static int e; /* DEFAULT_RAM(-1) */
static const int c4 = 14; /* MyConstSeg(1) */

#pragma CONST_SEG DEFAULT
static int f; /* DEFAULT_RAM(-1) */
static const int c5 = 15; /* DEFAULT_RAM(-1) */
```

---

#### Listing 8.42 Example of Segmentation with the -Cc Compiler Option

---

```
/* Placed into Segment: */
static int a; /* DEFAULT_RAM(-1) */
static const int c0 = 10; /* ROM_VAR(-2) */

#pragma DATA_SEG MyVarSeg
static int b; /* MyVarSeg(0) */
static const int c1 = 11; /* MyVarSeg(0) */

#pragma DATA_SEG DEFAULT
static int c; /* DEFAULT_RAM(-1) */
static const int c2 = 12; /* ROM_VAR(-2) */

#pragma DATA_SEG MyVarSeg
#pragma CONST_SEG MyConstSeg
static int d; /* MyVarSeg(0) */
static const int c3 = 13; /* MyConstSeg(1) */

#pragma DATA_SEG DEFAULT
static int e; /* DEFAULT_RAM(-1) */
static const int c4 = 14; /* MyConstSeg(1) */

#pragma CONST_SEG DEFAULT
static int f; /* DEFAULT_RAM(-1) */
static const int c5 = 15; /* ROM_VAR(-2) */
```

---

## Optimizations

The Compiler applies a variety of code-improving techniques under the term "optimization". This section provides a short overview about the most important optimizations.

## Peephole Optimizer

A peephole optimizer is a simple optimizer in a Compiler. A peephole optimizer tries to optimize specific code patterns on speed or code size. After recognizing these specific patterns, they will be replaced by other optimized patterns.

After code is generated by the backend of an optimizing Compiler, it is still possible that code patterns may result that are still capable of being optimized. The optimizations of the peephole optimizer are highly backend-dependent because the peephole optimizer was implemented with characteristic code patterns of the backend in mind.

Certain peephole optimizations only make sense in conjunction with other optimizations, or together with some code patterns. These patterns may have been generated by doing other optimizations. There are optimizations (e.g., removing of a branch to the next instructions) that are removed by the peephole optimizer, though they could have been removed by the branch optimizer as well. Such simple branch optimizations are performed in the peephole optimizer to reach new optimizable states.

## Strength Reduction

Strength reduction is an optimization that strives to replace expensive operations by cheaper ones, where the cost factor is either execution time or code size. Examples are the replacement of multiplication and division by constant powers of two with left or right shifts.

---

**NOTE** The compiler can only replace a division by two using a shift operation if either the target division is implemented the way that  $-1/2 == -1$ , or if the value to be divided is unsigned. The result is different for negative values. To give the compiler the possibility to use a shift, the C source code should already contain a shift, or the value to be shifted should be unsigned.

---

## Shift Optimizations

Shifting a byte variable by a constant number of bits is intensively analyzed. The Compiler always tries to implement such shifts in the most efficient way.

## Branch Optimizations

This optimization tries to minimize the span of branch instructions. The Compiler will never generate a long branch where a short branch would have sufficed. Also, branches to branches may be resolved into two branches to the same target. Redundant branches (e.g., a branch to the instruction immediately following it) may be removed.

## Dead-Code Elimination

The Compiler removes dead assignments while generating code. In some programs it may find additional cases of expressions that are not used.

## Constant-Variable Optimization

If a constant non-volatile variable is used in any expression, the Compiler replaces it by the constant value it holds. This needs less code than taking the object itself.

The constant non-volatile object itself is removed if there is no expression taking the address of it (take note of `ci` in Listing 8.43). This results in using less memory space.

### Listing 8.43 Example demonstrating constant-variable optimization

---

```
void f(void) {
    const int ci = 100; // ci removed (no address taken)
    const int ci2 = 200; // ci2 not removed (address taken below)
    const volatile int ci3 = 300; // ci3 not removed (volatile)
    int i;
    int *p;
    i = ci; // replaced by i = 100;
    i = ci2; // no replacement
    p = &ci2; // address taken
}
```

---

Global constant non-volatile variables are not removed. Their use in expressions are replaced by the constant value they hold.

Constant non-volatile arrays are also optimized (take note of `array[]` in Listing 8.44).

### Listing 8.44 Example demonstrating the optimization of a constant, non-volatile array

---

```
void g(void) {
    const int array[] = {1,2,3,4};
    int i;
    i = array[2]; // replaced by i=3;
}
```

---

## Tree Rewriting

The structure of the intermediate code between Frontend and Backend allows the Compiler to perform some optimizations on a higher level. Examples are shown in the following sections.



---

## Switch Statements

Efficient translation of switch statements is mandatory for any C Compiler. The Compiler applies different strategies, i.e., branch trees, jump tables, and a mixed strategy, depending on the case label values and their numbers. Table 8.6 describes how the Compiler implements these strategies.

**Table 8.6 Switch Implementations**

Method	Description
Branch Sequence	For small switches with scattered case label values, the Compiler generates an if ... elsif ... elsif ... else ... sequence if the Compiler switch -Os is active.
Branch Tree	For small switches with scattered case label values, the Compiler generates a branch tree. This is the equivalent to unrolling a binary search loop of a sorted jump table and therefore is very fast. However, there is a point at which this method is not feasible simply because it uses too much memory.
Jump Table	In such cases, the Compiler creates a table plus a call of a switch processor. There are two different switch processors. If there are a lot of labels with more or less consecutive values, a direct jump table is used. If the label values are scattered, a binary search table is used.
Mixed Strategy	Finally, there may be switches having "clusters" of label values separated by other labels with scattered values. In this case, a mixed strategy is applied, generating branch trees or search tables for the scattered labels and direct jump tables for the clusters.

## Absolute Values

Another example for optimization on a higher level is the calculation of absolute values. In C, the programmer has to write something on the order of:

```
float x, y;
```

```
x = (y < 0.0) ? -y : y;
```

This results in lengthy and inefficient code. The Compiler recognizes cases like this and treats them specially in order to generate the most efficient code. Only the most significant bit has to be cleared.

## Combined Assignments

The Compiler can also recognize the equivalence between the three following statements:

```
x = x + 1;  
x += 1;  
x++;
```

and between:

```
x = x / y;  
x /= y;
```

Therefore, the Compiler generates equally efficient code for either case.

## Using Qualifiers for Pointers

The use of qualifiers (const, volatile, ...) for pointers is confusing. This section provides some examples for the use of const or volatile as const and volatile are very common for Embedded Programming.

Consider the following example:

```
int i;  
const int ci;
```

The above definitions are: a 'normal' variable 'i' and a constant variable 'ci'. Each are placed into ROM. Note that for C++, the constant 'ci' must be initialized.

```
int *ip;  
const int *cip;
```

'ip' is a pointer to an 'int', where 'cip' is a pointer to a 'const int'.

```
int *const icp;  
const int *const cicp;
```

'icp' is a 'const pointer' to an 'int', where 'cicp' is a 'const pointer' to a 'const int'.

It helps if you know that the qualifier for such pointers is always on the right side of the '\*'. Another way is to read the source from right to left.

You can express this rule in the same way to volatile. Consider the following example of an 'array of five constant pointers to volatile integers':

```
volatile int *const arr[5];
```

'arr' is an array of five constant pointers pointing to volatile integers. Because the array itself is constant, it is put into ROM. It does not matter if the array is constant or not regarding where the pointers point to. Consider the next example:

```
const char *const *buf[] = {&a, &b};
```

Because the array of pointers is initialized, the array is not constant. 'buf' is a (non-constant) array of two pointers to constant pointers which points to constant characters. Thus 'buf' cannot be placed into ROM by the Compiler or Linker.

Consider a constant array of five ordinary function pointers. Assuming that:

```
void (*fp)(void);
```

is a function pointer 'fp' returning void and having void as parameter, you can define it with:

```
void (*fparr[5])(void);
```

It is also possible to use a typedef to separate the function pointer type and the array:

```
typedef void (*Func)(void);  
Func fp;  
Func fparr[5];
```

You can write a constant function pointer as:

```
void (*const cfp)(void);
```

Consider a constant function pointer having a constant int pointer as a parameter returning void:

```
void (*const cfp2)(int *const);
```

Or a const function pointer returning a pointer to a volatile double having two constant integers as parameter:

```
volatile double *(*const fp3)(const int, const int);
```

And one more:

```
void (*const fp[3])(void);
```

This is an array of three constant function pointers, having void as parameter and returning void. 'fp' is allocated in ROM because the 'fp' array is constant.

## ANSI-C Frontend

### Defining C Macros Containing HLI Assembler Code

---

Consider an example using function pointers:

```
int (* (** func0(int (*f) (void))) (int (*) (void))) (int (*)
(void)) {
    return 0;
}
```

It is actually a function called `func()`. This `func()` has one function pointer argument called `f`. The return value is more complicated in this example. It is actually a function pointer of a complex type. Here we do not explain where to put a `const` so that the destination of the returned pointer cannot be modified. Alternately, the same function is written more simply using `typedefs` (Listing 8.45):

#### Listing 8.45 Using typedefs

---

```
typedef int (*funcType1) (void);
typedef int (* funcType2) (funcType1);
typedef funcType2 (* funcType3) (funcType1);
funcType3* func0(funcType1 f) {
    return 0;
}
```

---

Now, the places of the `const` becomes obvious. Just behind the `*` in `funcType3`:

```
typedef funcType2 (* const constfuncType3) (funcType1);
constfuncType3* func1(funcType1 f) {
    return 0;
}
```

By the way, also in the first version here is the place where to put the `const`:

```
int (* (*const * func1(int (*f) (void))) (int (*) (void)))
(int (*) (void)) {
    return 0;
}
```

## Defining C Macros Containing HLI Assembler Code

You can define some ANSI C macros that contain HLI assembler statements when you are working with the HLI assembler. Because the HLI assembler is heavily Backend-dependent, the following example in Listing 8.46 uses a pseudo Assembler Language:

**Listing 8.46 Coding example**


---

```

CLR Reg0      ; Clear Register zero
CLR Reg1      ; Clear Register one
CLR var       ; Clear variable 'var' in memory
LOAD var,Reg0 ; Load the variable 'var' into Register 0
LOAD #0, Reg0 ; Load immediate value zero into Register 0
LOAD @var,Reg1 ; Load address of variable 'var' into Reg1
STORE Reg0,var ; Store Register 0 into variable 'var'

```

---

The HLI instructions are only used as a possible example. For real applications, you must replace the above pseudo HLI instructions with the HLI instructions for your target.

## Defining a Macro

An HLI assembler macro is defined by using the 'define' preprocessor directive.

For example, a macro could be defined to clear the R0 register. (Listing 8.47).

**Listing 8.47 Defining the ClearR0 macro.**


---

```

/* The following macro clears R0. */
#define ClearR0 {__asm CLR R0;}

```

---

The source code invokes the ClearR0 macro in the following manner (Listing 8.48).

**Listing 8.48 Invoking the ClearR0 macro.**


---

```

ClearR0;

```

---

And then the preprocessor expands the macro (Listing 8.49).

**Listing 8.49 Preprocessor expansion of ClearR0.**


---

```

{ __asm CLR R0 ; } ;

```

---

An HLI assembler macro can contain one or several HLI assembler instructions. As the ANSI-C preprocessor expands a macro on a single line, you cannot define an HLI assembler block in a macro. You can, however, define a list of HLI assembler instructions (Listing 8.50).

## ANSI-C Frontend

### Defining C Macros Containing HLI Assembler Code

---

#### Listing 8.50 Defining two macros on the same line of source code.

---

```
/* The following macro clears R0 and R1. */
#define ClearR0and1 {__asm CLR R0; __asm CLR R1; }
```

---

The macro is invoked in the following way in the source code (Listing 8.51).

#### Listing 8.51

---

```
ClearR0and1;
```

---

The preprocessor expands the macro:

```
{ __asm CLR R0 ; __asm CLR R1 ; } ;
```

You can define an HLI assembler macro on several lines using the line separator ‘\’.

---

**NOTE** This may enhance the readability of your source file. However, the ANSI-C preprocessor still expands the macro on a single line.

---

#### Listing 8.52 Defining a macro on more than one line of source code

---

```
/* The following macro clears R0 and R1. */
#define ClearR0andR1 {__asm CLR R0; \
                    __asm CLR R1;}

```

---

The macro is invoked in the following way in the source code (Listing 8.53).

#### Listing 8.53 Calling the ClearR0andR1 macro

---

```
ClearR0andR1;
```

---

The preprocessor expands the macro (Listing 8.54).

#### Listing 8.54 Preprocessor expansion of the ClearR0andR1 macro.

---

```
{__asm CLR R0; __asm CLR R1; };
```

---

## Using Macro Parameters

An HLI assembler macro may have some parameters which are referenced in the macro code. Listing 8.55 defines the Clear1 macro that uses the var parameter.

### Listing 8.55 Clear1 macro definition.

```
/* This macro initializes the specified variable to 0.*/  
#define Clear1(var) {__asm CLR var;}
```

---

Invoking the Clear1 macro in the source code:

```
Clear1(var1);
```

The preprocessor expands the Clear1 macro

```
{__asm CLR var1 ; };
```

## Using the Immediate-Addressing Mode in HLI Assembler Macros

There may be one ambiguity if you are using the immediate addressing mode inside of a macro.

For the ANSI-C preprocessor, the symbol # inside of a macro has a specific meaning (string constructor).

Using #pragma NO\_STRING\_CONSTR: No String Concatenation during Preprocessing, the Compiler is instructed that in all the macros defined afterward, the instructions should remain unchanged wherever the symbol # is specified. This macro is valid for the rest of the file in which it is specified.

### Listing 8.56 Definition of the Clear2 macro

---

```
/* This macro initializes the specified variable to 0.*/  
#pragma NO_STRING_CONSTR  
#define Clear2(var) {__asm LOAD #0,Reg0;__asm STORE Reg0,var;}
```

---

Invoking the Clear2 macro in the source code

```
Clear2(var1);
```

## ANSI-C Frontend

### Defining C Macros Containing HLI Assembler Code

---

The preprocessor expands the Clear2 macro

```
{ __asm LOAD #0,Reg0;__asm STORE Reg0,var1; };
```

## Generating Unique Labels in HLI Assembler Macros

When some labels are defined in HLI Assembler Macros, if you invoke the same macro twice in the same function, the ANSI C preprocessor generates the same label twice (once in each macro expansion). Use the special string concatenation operator of the ANSI-C preprocessor ('##') in order to generate unique labels. See Listing 8.57.

### Listing 8.57 Using the ANSI C preprocessor string concatenation operator

---

```
/* The following macro copies the string pointed to by 'src'
   into the string pointed to by 'dest'.
   'src' and 'dest' must be valid arrays of characters.
   'inst' is the instance number of the macro call. This
   parameter must be different for each invocation of the
   macro to allow the generation of unique labels. */
#pragma NO_STRING_CONSTR
#define copyMacro2(src, dest, inst) { \
__asm          LOAD   @src,Reg0; /* load src addr */ \
__asm          LOAD   @dest,Reg1; /* load dst addr */ \
__asm          CLR    Reg2;      /* clear index reg */ \
__asm lp##inst: LOADB (Reg2, Reg0); /* load byte reg indir */ \
__asm          STOREB (Reg2, Reg1); /* store byte reg indir */ \
__asm          ADD    #1,Reg2; /* increment index register */ \
__asm          TST    Reg2;      /* test if not zero */ \
__asm          BNE    lp##inst; }
```

---

Invoking the copyMacro2 macro in the source code:

```
copyMacro2(source2, destination2, 1);
copyMacro2(source2, destination3, 2);
```

During expansion of the first macro, the preprocessor generates an 'lp1' label. During expansion of the second macro, an 'lp2' label is created.



## Generating Assembler Include Files (-La compiler option)

In many projects it often makes sense to use both a C compiler and an assembler. Both have different advantages. The compiler uses portable and readable code while the assembler provides full control for time-critical applications, or for direct accessing of the hardware.

The compiler cannot read include files of the assembler, and the assembler cannot read the header files of the compiler.

The assembler's include file output of the compiler lets both tools use one single source to share constants, variables or labels, and even structure fields.

The compiler writes an output file in the format of the assembler which contains all information needed of a C header file.

The current implementation supports the following mappings:

- Macros  
C defines are translated to assembler EQU directives.
- enum values  
C enum values are translated to EQU directives.
- C types  
The size of any type and the offset of structure fields is generated for all typedefs. For bitfield structure fields, the bit offset and the bit size are also generated.
- Functions  
For each function an XREF entry is generated.
- Variables  
C Variables are generated with an XREF. In addition, for structures or unions all fields are defined with an EQU directive.
- Comments  
C style comments (*/\* ... \*/*) are included as assembler comments (*;(....)*).

### General

A header file must be specially prepared to generate the assembler include file ().

A pragma anywhere in the header file can enable assembler output:

```
#pragma CREATE_ASM_LISTING ON
```

## ANSI-C Frontend

### Defining C Macros Containing HLI Assembler Code

---

Only macro definitions and declarations behind this pragma are generated. The compiler stops generating future elements when `#pragma CREATE_ASM_LISTING`: Create an Assembler Include File Listing occurs with an `OFF` parameter.

```
#pragma CREATE_ASM_LISTING OFF
```

Not all entries generate legal assembler constructs. Care must be taken for macros. The compiler does not check for legal assembler syntax when translating macros. Macros containing elements not supported by the assembler should be in a section controlled by `"#pragma CREATE_ASM_LISTING OFF"`.

The compiler only creates an output file when the `-La` option is specified and the compiled sources contain `#pragma CREATE_ASM_LISTING ON` (Listing 8.58).

#### Listing 8.58 Header file: a.h

---

```
#pragma CREATE_ASM_LISTING ON
typedef struct {
    short i;
    short j;
} Struct;
Struct Var;
void f(void);
#pragma CREATE_ASM_LISTING OFF
```

---

When the compiler reads this header file with the `-La=a.inc a.h` option, it generates the following (Listing 8.59).

#### Listing 8.59 a.inc file

---

```
Struct_SIZE    EQU $4
Struct_i       EQU $0
Struct_j       EQU $2
               XREF Var
Var_i          EQU Var + $0
Var_j          EQU Var + $2
               XREF f
```

---

You can now use the assembler `INCLUDE` directive to include this file into any assembler file. The content of the C variable, `Var_i`, can also be accessed from the assembler without any uncertain assumptions about the alignment used by the compiler. Also, whenever a field is added to the structure `Struct`, the assembler code must not be altered. You must, however, regenerate the `a.inc` file with a make tool.

Usually the assembler include file is not created every time the compiler reads the header file. It is only created in a separate pass when the header file has changed significantly.

The `-La` option is only specified when the compiler must generate `a.inc`. If `-La` is always present, `a.inc` is always generated. A make tool will always restart the assembler because the assembler files depend on `a.inc`. Such a makefile might be similar to Listing 8.60:

---

### Listing 8.60 Sample makefile

---

```
a.inc : a.h
    $(CC) -La=a.inc a.h
a_c.o : a_c.c a.h
    $(CC) a_c.c
a_asm.o : a_asm.asm a.inc
    $(ASM) a_asm.asm
```

---

The order of elements in the header file is the same as the order of the elements in the created file, except that comments may be inside of elements in the C file. In this case, the comments may be before or after the whole element.

The order of defines does not matter for the compiler. The order of `EQU` directives does matter for the assembler. If the assembler has problems with the order of `EQU` directives in a generated file, the corresponding header file must be changed accordingly.

## Macros

The translation of defines is done lexically and not semantically. So the compiler does not check the accuracy of the define.

The following example (Listing 8.61) shows some uses of this feature:

---

### Listing 8.61 Example Source code

---

```
#pragma CREATE_ASM_LISTING ON
int i;
#define UseI i
#define Constant 1
#define Sum Constant+0X1000+01234
```

---

The source code in Listing 8.61 produces the following output (Listing 8.62):

---

### Listing 8.62 Disassembly listing of Listing 8.61

---

```
UseI                XREF    i
                    EQU     i
```

---

## ANSI-C Frontend

### Defining C Macros Containing HLI Assembler Code

---

```
Constant          EQU    1
Sum               EQU    Constant + $1000 + @234
```

---

The hexadecimal C constant `0x1000` was translated to `$1000` while the octal `01234` was translated to `@1234`. In addition, the compiler has inserted one space between every two tokens. These are the only changes the compiler makes in the assembler listing for defines.

Macros with parameters, predefined macros, and macros with no defined value are not generated.

The following defines (Listing 8.63) do not work or are not generated:

#### Listing 8.63 Improper defines

---

```
#pragma CREATE_ASM_LISTING ON
int i;
#define AddressOfI &i
#define ConstantInt ((int)1)
#define Mul7(a) a*7
#define Nothing
#define useUndef UndefFkt*6
#define Anything § § / % & % / & + * % ç 65467568756 86
```

---

The source code in Listing 8.63 produces the following output (Listing 8.64):

#### Listing 8.64 Disassembly listing of Listing 8.63

---

```
AddressOfI      XREF  i
ConstantInt    EQU   & i
useUndef       EQU   ( ( int ) 1 )
Anything       EQU   UndefFkt * 6
Anything       EQU   § § / % & % / & + * % ç 65467568756 86
```

---

The `AddressOfI` macro does not assemble because the assembler does not know to interpret the `&` C address operator. Also, other C-specific operators such as dereferenciation (`*ptr`) must not be used. The compiler generates them into the assembler listing file without any translation.

The `ConstantInt` macro does not work because the assembler does not know the cast syntax and the types.

Macros with parameters are not written to the listing,. Therefore, `Mul7` does not occur in the listing. Also, macros just defined with no actual value as `Nothing` are not generated.

The C preprocessor does not care about the syntactical content of the macro, though the assembler EQU directive does. Therefore, the compiler has no problems with the useUndef macro using the undefined UndefFkt object. The assembler EQU directive requires that all used objects are defined.

The Anything macro shows that the compiler does not care about the content of a macro. The assembler, of course, cannot treat these random characters.

These types of macros are in a header file used to generate the assembler include file. They must only be in a region started with "#pragma CREATE\_ASM\_LISTING OFF" so that the compiler will not generate anything for them.

## Enumerations

An enum in C has a unique name and a defined value (Listing 8.65).

### Listing 8.65 Enumerations

---

```
#pragma CREATE_ASM_LISTING ON
enum {
    E1=4,
    E2=47,
    E3=-1*7
};
```

---

They are simply generated by the compiler as an EQU directive (Listing 8.66).

### Listing 8.66 Disassembly of Listing 8.65

---

```
E1          EQU $4
E2          EQU $2F
E3          EQU $FFFFFFFF9
```

---



---

**NOTE** Negative values are generated as 32-bit hex numbers.

---

## Types

As it does not make sense to generate the size of any occurring type, only typedefs are considered.

The size of the newly defined type is specified for all typedefs.

## ANSI-C Frontend

### Defining C Macros Containing HLI Assembler Code

---

#### Listing 8.67 typedefs

---

```
#pragma CREATE_ASM_LISTING ON
typedef long LONG;
struct tagA {
    char a;
    short b;
};
typedef struct {
    long d;
    struct tagA e;
    int f:2;
    int g:1;
} str;
```

---

For the name for the size of a typedef, an additional term "\_SIZE" is appended to the end of the typedef's name. For structures, the offset of all structure fields is generated relative to the structure's start. The names of the structure offsets are generated by appending the structure field's name after an underline ("\_") to the typedef's name (Listing 8.68).

#### Listing 8.68 Disassembly of Listing 8.67

---

LONG_SIZE	EQU \$4
str_SIZE	EQU \$8
str_d	EQU \$0
str_e	EQU \$4
str_e_a	EQU \$4
str_e_b	EQU \$5
str_f	EQU \$7
str_f_BIT_WIDTH	EQU \$2
str_f_BIT_OFFSET	EQU \$0
str_g	EQU \$7
str_g_BIT_WIDTH	EQU \$1
str_g_BIT_OFFSET	EQU \$2

---

All structure fields inside of another structure are contained within that structure. The generated name contains all the names for all fields listed in the path. If any element of the path does not have a name (e.g., an anonymous union), this element is not generated.

The width and the offset are also generated for all bitfield members. The offset 0 specifies the least significant bit, which is accessed with mask 0x1. The offset 2 the most significant bit, which is accessed with mask 0x4. The width specifies the number of bits.

The offsets, bit widths and bit offsets, given here are examples. Different compilers may generate different values. In C, the structure alignment and the bitfield allocation is determined by the compiler which specifies the correct values.

## Functions

Declared functions are generated by the `XREF` directive. This enables them to be used with the Assembler. The function to be called from C, but defined in assembly code, should not be generated into the output file as the Assembler does not allow the redefinition of labels declared with `XREF`. Such function prototypes are placed in an area started with "`#pragma CREATE_ASM_LISTING OFF`", as shown in Listing 8.69.

### Listing 8.69 Function prototyping with the `CREATE_ASM_LISTING` pragma

---

```
#pragma CREATE_ASM_LISTING ON
void main(void);
void f_C(int i, long l);

#pragma CREATE_ASM_LISTING OFF
void f_asm(void);
```

---

The source code in the above Listings disassembles as in Listing 8.70.

### Listing 8.70 Disassembly of Listing 8.69

---

```
XREF main
XREF f_C
```

---

## Variables

An example of variables is shown in Listing 8.71.

### Listing 8.71 Examples of variables

---

```
#pragma CREATE_ASM_LISTING ON
struct A {
    char a;
    int i:2;
};
struct A VarA;
#pragma DATA_SEG __SHORT_SEG ShortSeg
```

---

## ANSI-C Frontend

### Defining C Macros Containing HLI Assembler Code

---

```
int VarInt;
```

---

Variables are declared with `XREF`. In addition, for structures, every field is defined with an `EQU` directive. For bitfields, the bit offset and bit size are also defined.

Variables in the `__SHORT_SEG` segment are defined with `XREF .B` to inform the assembler about the direct access. Fields in structures in `__SHORT_SEG` segments, are defined with a `EQU .B` directive.

#### Listing 8.72 Disassembly of Listing 8.71

---

```
VarA_a          XREF VarA
VarA_a          EQU VarA + $0
VarA_i          EQU VarA + $1
VarA_i_BIT_WIDTH EQU $2
VarA_i_BIT_OFFSET EQU $0
VarA_i_BIT_OFFSET XREF .B VarInt
```

---

The variable size is not explicitly written. To generate the variable size, use a typedef with the variable type.

The offsets, bit widths and bit offsets, given here are examples. Different compilers may generate different values. In C, the structure alignment and the bitfield allocation is determined by the compiler which specifies the correct values.

## Comments

Comments inside a region generated with `#pragma CREATE_ASM_LISTING ON` are also written on a single line in the assembler include file.

Comments inside of a typedef, a structure, or a variable declaration are placed either before or after the declaration. They are never placed inside the declaration, even if the declaration contains multiple lines. Therefore, a comment after a structure field in a typedef is written before or after the whole typedef, not just after the type field. Every comment is on a single line. An empty comment (`/* */`) inserts an empty line into the created file. See Listing 8.73 for an example using constants.

#### Listing 8.73 Example using comments

---

```
#pragma CREATE_ASM_LISTING ON
/*
The function main is called by the startup code.
The function is written in C. Its purpose is
to initialize the application. */
void main(void);
```

---



```
/*
   The SIZEOF_INT macro specified the size of an integer type
   in the compiler. */
typedef int SIZEOF_INT;
#pragma CREATE_ASM_LISTING OFF
```

---

When disassembled, the listing above is as shown is Listing 8.74.

#### Listing 8.74 Disassembly of Listing 8.73

---

```
; The function main is called by the startup code.
; The function is written in C. Its purpose is
; to initialize the application.
           XREF main
;
;   The SIZEOF_INT macro specified the size of an integer type
;   in the compiler.
SIZEOF_INT_SIZE      EQU $2
```

---

## Guidelines

The `-La` option translates specified parts of header files into an include file to import labels and defines into an assembler source. Because the `-La` compiler option is very powerful, its incorrect use must be avoided using the following guidelines implemented in a real project. This section describes how the programmer uses this option to combine C and assembler sources, both using common header files.

The following general implementation recommendations help to avoid problems when writing software using the common header file technique.

- All interface memory reservations or definitions must be made in C source files. Memory areas, only accessed from assembler files, can still be defined in the common assembler manner.
- Compile only C header files (and not the C source files) with the `-La` option to avoid multiple defines and other problems. The project-related makefile must contain an inference rules section that defines the C header file(s)-dependent include files to be created.
- Use `#pragma CREATE_ASM_LISTING ON/OFF` only in C header files. This `#pragma` selects the objects which should be translated to the assembler include file. The created assembler include file then holds the corresponding assembler directives.

## ANSI-C Frontend

### Defining C Macros Containing HLI Assembler Code

---

- The `-La` option should not be part of the command line options used for all compilations. Use this option in combination with the `-Cx` (no Code Generation) option. Without this option, the compiler creates an object file which could accidentally overwrite a C source object file.
- Remember to extend the list of dependencies for assembler sources in your make file.
- Check if the compiler-created assembler include file is included into your assembler source.

---

**NOTE** In case of a zero-page declared object (if this is supported by the target), the compiler translates it into an `XREF .B` directive for the base address of a variable or constant. The compiler translates structure fields in the zero page into an `EQU.B` directive in order to access them. Explicit zero-page addressing syntax may be necessary as some assemblers use extended addresses to `EQU.B` defined labels.

Project-defined data types must be declared in the C header file by including a global project header (e.g., `global.h`). This is necessary as the header file is compiled in a standalone fashion.

---

# Generating Compact Code

---

The Compiler tries whenever possible to generate compact and efficient code. But not everything is handled directly by the Compiler. With a little help from the programmer, it is possible to reach denser code. Some Compiler options, or using `__SHORT_SEG` segments (if available), help to generate compact code.

## Compiler Options

Using the following compiler options helps to reduce the size of the code generated. Note that not all options may be available for each target.

### **-Or: Register Optimization**

When accessing pointer fields, this option prevents the compiler from reloading the address of the pointer for each access. An index register holds the pointer value over statements where possible.

---

**NOTE** This option may not be available for all targets.

---

### **-Oi: Inlining: Inline Functions**

Use the inline keyword or the command line option `-Oi` for C/C++ functions. Defining a function before it is used helps the Compiler to inline it:

---

```

/* OK */
void foo(void);
void main(void) {
    foo();
}
void foo(void) {
    // ...
}

/* better! */
void foo(void) {
    // ...
}
void main(void) {
    foo();
}

```

---

This also helps the compiler to use a relative branch instruction instead an absolute.

# `__SHORT_SEG` Segments

Variables allocated on the direct page (between 0 and 0xFF) are accessed using the direct addressing mode. The Compiler will allocate some variables on the direct page if they are defined in a `__SHORT_SEG` segment (Listing 9.1).

## Listing 9.1 Allocate frequently-used variables on the direct page

---

```
#pragma DATA_SEG __SHORT_SEG myShortSegment
unsigned int myVar1, myVar2;
#pragma DATA_SEG DEFAULT
unsigned int myvar3, myVar4
```

---

In the previous example, 'myVar1' and 'myVar2' are both accessed using the direct addressing mode. Variables 'myVar3' and 'myVar4' are accessed using the extended addressing mode.

When some exported variables are defined in a `__SHORT_SEG` segment, the external declaration for these variables must also specify that they are allocated in a `__SHORT_SEG` segment. The External definition of the variable defined above looks like:

```
#pragma DATA_SEG __SHORT_SEG myShortSegment
extern unsigned int myVar1, myVar2;
#pragma DATA_SEG DEFAULT
extern unsigned int myvar3, myVar4
```

The segment must be placed on the direct page in the PRM file (Listing 9.2).

## Listing 9.2 Linker parameter file

---

```
LINK test.abs
NAMES test.o startup.o ansi.lib END
SECTIONS
    Z_RAM = READ_WRITE 0x0080 TO 0x00FF;
    MY_RAM = READ_WRITE 0x0100 TO 0x01FF;
    MY_ROM = READ_ONLY 0xF000 TO 0xFEFF;
PLACEMENT
    DEFAULT_ROM INTO MY_ROM;
    DEFAULT_RAM INTO MY_RAM;
    _ZEROPAGE, myShortSegment INTO Z_RAM;
END
STACKSIZE 0x60
```

---

```
VECTOR 0 _Startup /* set reset vector on _Startup */
```

---

---

**NOTE** The linker is case-sensitive. The segment name must be identical in the C and PRM files.

---

## Defining I/O Registers

The I/O Registers are usually based at address 0. In order to tell the compiler it must use direct addressing mode to access the I/O registers, these registers are defined in a `__SHORT_SEG` section (if available) based at the specified address.

The I/O register is defined in the C source file as in Listing 9.3.

### Listing 9.3 Definition of an I/O Register

---

```
typedef struct {
    unsigned char SCC1;
    unsigned char SCC2;
    unsigned char SCC3;
    unsigned char SCS1;
    unsigned char SCS2;
    unsigned char SCD;
    unsigned char SCBR;
} SCIStruct;
#pragma DATA_SEG __SHORT_SEG SCISegs
SCIStruct SCI;
#pragma DATA_SEG DEFAULT
```

---

Then the segment must be placed at the appropriate address in the PRM file (Listing 9.4).

### Listing 9.4 Linker parameter file Allocating the I/O Register

---

```
LINK test.abs
NAMES test.o startup.o ansi.lib END
SECTIONS
    SCI_RG = READ_WRITE 0x0013 TO 0x0019;
    Z_RAM = READ_WRITE 0x0080 TO 0x00FF;
    MY_RAM = READ_WRITE 0x0100 TO 0x01FF;
    MY_ROM = READ_ONLY 0xF000 TO 0xFEFF;
PLACEMENT
    DEFAULT_ROM INTO MY_ROM;
    DEFAULT_RAM INTO MY_RAM;
```

---

## Generating Compact Code

### Programming Guidelines

---

```
_ZEROPAGE                INTO  Z_RAM;
SCIRegs                   INTO  SCI_RG;
END
STACKSIZE 0x60
VECTOR 0 _Startup /* set reset vector on _Startup */
```

---

---

**NOTE** The linker is case-sensitive. The segment name must be identical in the C/C++ and PRM files.

---

## Programming Guidelines

Following a few programming guidelines helps to reduce code size. Many things are optimized by the Compiler. However, if the programming style is very complex or if it forces the Compiler to perform special code sequences, code efficiency is not would be expected from a typical optimization.

### Constant Function at a Specific Address

Sometimes functions are placed at a specific address, but the sources or information regarding them are not available. The programmer knows that the function starts at address 0x1234 and wants to call it. Without having the definition of the function, the program runs into a linker error due to the lack of the target function code. The solution is to use a constant function pointer:

```
void (*const fktPtr)(void) = (void(*) (void))0x1234;
void main(void) {
    fktPtr();
}
```

This gives you efficient code and no linker errors. However, it is necessary that the function at 0x1234 really exists.

Even a better way (without the need for a function pointer):

```
#define erase ((void(*) (void)) (0xfc06))
void main(void) {
    erase(); /* call function at address 0xfc06 */
}
```

### HLI Assembly

Do not mix High-level Inline (HLI) Assembly with C declarations and statements (see Listing 9.5). Using HLI assembly may affect the register trace of the compiler. The

Compiler cannot touch HLI Assembly, and thus it is out of range for any optimizations (except branch optimization, of course).

**Listing 9.5 Mixing HLI Assembly with C Statements (not recommended).**

---

```
void foo(void) {
    /* some local variable declarations */
    /* some C/C++ statements */
    __asm {
        /* some HLI statements */
    }
    /* maybe other C/C++ statements */
}
```

---

The Compiler in the worst case has to assume that everything has changed. It cannot hold variables into registers over HLI statements. Normally it is better to place special HLI code sequences into separate functions. However, there is the drawback of an additional call or return. Placing HLI instructions into separate functions (and module) simplifies porting the software to another target (Listing 9.6).

**Listing 9.6 HLI Statements are not mixed with C Statements (recommended).**

---

```
/* hardware.c */
void special_hli(void) {
    __asm {
        /* some HLI statements */
    }
}
/* foo.c */
void foo(void) {
    /* some local variable declarations */
    /* some C/C++ statements */
    special_hli();
    /* maybe other C/C++ statements */
}
```

---

## Post and Pre Operators in Complex Expressions

Writing a complex program results in complex code. In general it is the job of the compiler to optimize complex functions. Some rules may help the compiler to generate efficient code.

## Generating Compact Code

### Programming Guidelines

---

If the target does not support powerful postincrement or postdecrement and preincrement or predecrement instructions, it is not recommended to use the ‘++’ and ‘--’ operator in complex expressions. Especially postincrement or postdecrement may result in additional code:

```
a[i++] = b[--j];
```

Write the above statement as:

```
j--; a[i] = b[j]; i++;
```

Using it in simple expressions as:

```
i++;
```

Avoid assignments in parameter passing or side effects (as ‘++’ and ‘--’). The evaluation order of parameters is undefined (ANSI-C standard 6.3.2.2) and may vary from Compiler to Compiler, and even from one release to another:

## Example

```
i = 3;
foo(i++, --i);
```

In the above example, `foo()` is called either with ‘`foo(3, 3)`’ or with ‘`foo(2, 2)`’.

## Boolean Types

In C, the boolean type of an expression is an ‘int’. A variable or expression evaluating to ‘0’ (zero) is FALSE and everything else (!= 0) is TRUE. Instead of using an ‘int’ (usually 16 or 32 bits), it may be better to use an 8-bit type to hold a boolean result. For ANSI-C compliance, the basic boolean types are declared in ‘`stdtypes.h`’:

```
typedef int Bool;
#define TRUE 1
#define FALSE 0
```

Using:

```
typedef Byte Bool_8;
```

from ‘`stdtypes.h`’ (‘Byte’ is an unsigned 8-bit data type also declared in ‘`stdtypes.h`’) reduces memory usage and improves code density.



## printf() and scanf()

The `printf()` or `scanf()` code in the ANSI library can be reduced if no floating point support (`%f`) is used. Refer to the ANSI library reference and `printf.c` or `scanf.c` in your library for details on how to save code (not using `float` or `doubles` in `printf()` may result in half the code).

## Bitfields

Using bitfields to save memory may be a bad idea as bitfields produce a lot of additional code. For ANSI-C compliance, bitfields have a type of 'signed int', thus a bitfield of size 1 is either '-1' or '0'. This could force the compiler to 'sign extend' operations:

```
struct {
    int b:0; /* -1 or 0 */
} B;
int i = B.b; /* load the bit, sign extend it to -1 or 0 */
```

Sign extensions are normally time- and code-inefficient operations.

## Struct Returns

Normally the compiler has first to allocate space on the stack for the return value (1) and then to call the function (2). Phase (3) is for copying the return value to the variable `s`. In the callee 'foo' during the return sequence, the Compiler has to copy the return value (4, struct copy).

Depending on the size of the struct, this may be done inline. After return, the caller 'main' has to copy the result back into 's'. Depending on the Compiler or Target, it is possible to optimize some sequences (avoiding some copy operations). However, returning a struct by value may use a lot of execution time, and this could mean a lot of code and stack usage.

### Listing 9.7 Returning a struct can force the Compiler to produce lengthy code.

---

```
struct S foo(void)

    /* ... */
    return s; // (4)
}

void main(void) {
    struct S s;
    /* ... */
    s = foo(); // (1), (2), (3)
```

## Generating Compact Code

### Programming Guidelines

---

```
/* ... */  
}
```

---

With the example in Listing 9.8, the Compiler just has to pass the destination address and to call 'foo' (2). On the callee side, the callee copies the result indirectly into the destination (4). This approach reduces stack usage, avoids copying structs, and results in denser code. Note that the Compiler may also inline the above sequence (if supported). But for rare cases the above sequence may not be exactly the same as returning the struct by value (e.g., if the destination struct is modified in the callee).

#### Listing 9.8 A better way is to pass only a pointer to the callee for the return value.

---

```
void foo(struct S *sp) {  
    /* ... */  
    *sp = s; // (4)  
}  
void main(void) {  
    S s;  
    /* ... */  
    foo(&s); // (2)  
    /* ... */  
}
```

---

## Local Variables

Using local variables instead of global variable results in better manageability of the application as side effects are reduced or totally avoided. Using local variables or parameters reduces global memory usage but increases stack usage.

Stack access capabilities of the target influences the code quality. Depending on the target capabilities, access to local variables may be very inefficient. A reason might be the lack of a dedicated stack pointer (another address register has to be used instead, thus it might not be used for other values) or access to local variables is inefficient due the target architecture (limited offsets, only few addressing modes).

Allocating a huge amount of local variables may be inefficient because the Compiler has to generate a complex sequence to allocate the stack frame in the beginning of the function and to deallocate them in the exit part (Listing 9.9):

#### Listing 9.9 Function with a huge amount of local variables

---

```
void foo(void) {  
    /* huge amount of local variables: allocate space! */  
    /* ... */  
    /* deallocate huge amount of local variables */  
}
```

---

If the target provides special entry or exit instructions for such cases, allocation of many local variables is not a problem. A solution is to use global or static local variables. This deteriorates maintainability and also may waste global address space.

The Compiler may offer an option to overlap parameter or local variables using a technique called ‘overlapping’. Local variables or parameters are allocated as globals. The linker overlaps them depending on their use. For targets with limited stack (e.g., no stack addressing capabilities), this often is the only solution. However this solution makes the code non-reentrant (no recursion is allowed).

## Parameter Passing

Avoid parameters which exceed the data passed through registers (see Backend).

## Unsigned Data Types

Using unsigned data types is acceptable as signed operations are much more complex than unsigned ones (e.g., shifts, divisions and bitfield operations). But it is a bad idea to use unsigned types just because a value is always larger or equal to zero, and because the type can hold a larger positive number.

## Inlining and Macros

### abs() and labs()

Use the corresponding macro `M_ABS` defined in `stdlib.h` instead of calling `abs()` and `labs()` in the `stdlib`:

```
/* extract
/* macro definitions of abs() and labs() */
#define M_ABS(j) (((j) >= 0) ? (j) : -(j))
extern int      abs(int j);
extern long int labs(long int j);
```

But be careful as `M_ABS()` is a macro,

```
i = M_ABS(j++);
```

and is not the same as:

```
i = abs(j++);
```

## **memcpy() and memcpy2()**

ANSI-C requires that the `memcpy()` library function in `'strings.h'` returns a pointer of the destination and handles and is able to also handle a count of zero:

### **Listing 9.10 Excerpts from the `string.h` and `string.c` files relating to `memcpy()`**

---

```
/* extract of string.h *
extern void * memcpy(void *dest, const void * source, size_t count);

extern void  memcpy2(void *dest, const void * source, size_t count);
/* this function does not return dest and assumes count > 0 */

/* extract of string.c */
void * memcpy(void *dest, const void *source, size_t count) {
    uchar *sd = dest;
    uchar *ss = source;

    while (count--)
        *sd++ = *ss++;

    return (dest);
}
```

---

If the function does not have to return the destination and it does have to handle a count of zero, the `memcpy2()` function in Listing 9.11 is much simpler and faster:

### **Listing 9.11 Excerpts from the `string.c` File relating to `memcpy2()`**

---

```
/* extract of string.c */
void
memcpy2(void *dest, const void* source, size_t count) {
    /* this func does not return dest and assumes count > 0 */
    do {
        *((uchar *)dest)++ = *((uchar*)source)++;
    } while(count--);
}
```

---

Replacing calls to `memcpy()` with calls to `memcpy2()` saves runtime and code size.

## **Data Types**

Do not use larger data types than necessary. Use IEEE32 floating point format both for float and doubles if possible. Set the `enum` type to a smaller type than `'int'` using the `-T` option. Avoid data types larger than registers.

## Short Segments

Whenever possible and available (not all targets support it), place frequently used global variables into a `DIRECT` or `__SHORT_SEG` segment using

```
#pragma DATA_SEG __SHORT_SEG MySeg
```

## Qualifiers

Use the `'const'` qualifier to help the compiler. The `'const'` objects are placed into ROM for the HIWARE object-file format if the `-Cc` compiler option is given.

**Generating Compact Code**  
*Programming Guidelines*

---

# HC(S)08 Backend

---

The Backend is the target-dependent part of a Compiler containing the code generator. This chapter discusses the technical details of the Backend for the M68HC(S)08 family.

## Memory Models

This section describes the following memory models:

- SMALL Model
- TINY Model

### SMALL Model

The SMALL memory model is the default. All pointers and functions are assumed to have 16-bit addresses if not explicitly specified. In the SMALL memory model, code and data must be in the 64 kB address space. The SMALL memory model is selected with the `-Ms` compiler option, `-M` (`-Ms`, `-Mt`): Memory Model.

### TINY Model

In the TINY memory model, all data including stack must fit into the zero page. Data pointers are assumed to have 8-bit addresses if not explicitly specified with the keyword `__far`. The code address space is still 64 kB and function pointers are still 16-bit large. The TINY memory model is selected with the `-Mt` compiler option.

## Non-ANSI Keywords

Table 10.1 gives an overview of the supported non-ANSI keywords.

**Table 10.1 Supported non-ANSI Keywords**

Keyword	Supported For		
	Data Pointer	Function Pointer	Function
<code>__far</code>	yes	no	no
<code>__near</code>	yes	no	no
<code>interrupt</code>	no	no	yes

## Data Types

This section describes how the basic types of ANSI-C are implemented by the MC68HC08 Backend.

### Scalar Types

All basic types may be changed with the `-T`: Flexible Type Management compiler option. All scalar types (except `char`) are without a signed/unsigned qualifier, and their default values are signed (e.g., `int` is the same as `signed int`).

Table 10.2 gives the sizes of the simple types together with the possible formats using the `-T` option.

**Table 10.2 Types and Formats for the `-T` compiler option**

Type	Default Format	Default Value Range		Formats available with the <code>-T</code> Option
		Min	Max	
<code>char (unsigned)</code>	8-bit	0	255	8-bit, 16-bit, 32-bit
<code>signed char</code>	8-bit	-128	127	8-bit, 16-bit, 32-bit
<code>unsigned char</code>	8-bit	0	255	8-bit, 16-bit, 32-bit
<code>signed short</code>	16-bit	-32768	32767	8-bit, 16-bit, 32-bit
<code>unsigned short</code>	16-bit	0	65535	8-bit, 16-bit, 32-bit
<code>enum (signed)</code>	16-bit	-32768	32767	8-bit, 16-bit, 32-bit



**Table 10.2 Types and Formats for the -T compiler option (continued)**

Type	Default Format	Default Value Range		Formats available with the -T Option
		Min	Max	
signed int	16-bit	-32768	32767	8-bit, 16-bit, 32-bit
unsigned int	16-bit	0	65535	8-bit, 16-bit, 32-bit
signed long	32-bit	-2147483648	2147483647	8-bit, 16-bit, 32-bit
unsigned long	32-bit	0	4294967295	8-bit, 16-bit, 32-bit
signed long long	32-bit	-2147483648	2147483647	8-bit, 16-bit, 32-bit
unsigned long long	32-bit	0	4294967295	8-bit, 16-bit, 32-bit

**NOTE** Plain type char is unsigned. This default can be changed by the -T option.

## Floating-Point Types

The Compiler supports the two IEEE standard formats (32 and 64 bits wide) for floating-point types. Table 10.3 shows the range of values for the various floating-point representations.

The default format for a `float` is implemented as a 32-bit IEEE, whereas a `double` is in IEEE 64-bit format. If you need speed more than the added accuracy of `double` arithmetic operations, issue the `-Fd: Doubles are IEEE32` command-line option. If this option is used, both `float` and `double` are implemented using the IEEE 32-bit format.

The `-T: Flexible Type Management` option is used to change the default format of a `float` or `double`.

**Table 10.3 Floating-Point representation**

Type	Default Format	Default Value Range		Formats available with -T
		Min	Max	
float	IEEE32	1.17549435E-38F	3.402823466E+38F	IEEE32, IEEE64
double	IEEE64	2.2259738585972014E-308	1.7976931348623157E+308	IEEE32, IEEE64

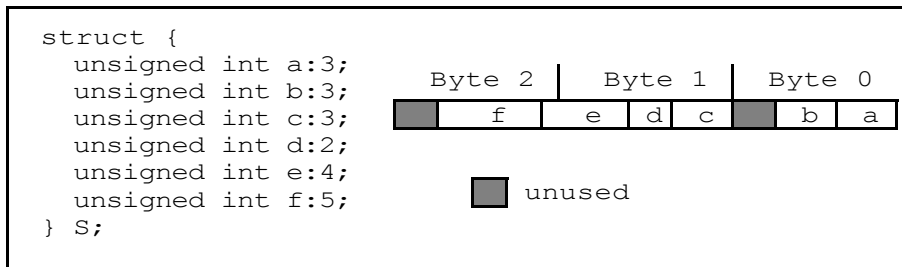
**Table 10.3 Floating-Point representation (continued)**

Type	Default Format	Default Value Range		Formats available with -T
		Min	Max	
long double	IEEE64	2.2259738585972014E-308	1.7976931348623157E+308	IEEE32, IEEE64
long long double	IEEE64	2.2259738585972014E-308	1.7976931348623157E+308	IEEE32, IEEE64

## Bitfields

The maximum width of bitfields is 16 bits. The allocation unit is a byte. The Compiler uses words only if a bitfield is wider than eight bits, or using bytes would cause more than two bits to be left unused. Allocation order is from the least significant bit up to the most significant bit in the order of declaration. The following figure illustrates an allocation scheme.

**Figure 10.1 Allocation of six bitfields**



The following example demonstrates a simple C code source with various ways to access bitfields, together with the produced code:

**Listing 10.1 Demonstration of bitfield instructions for the HC08 Compiler**

```

#pragma DATA_SEG __SHORT_SEG _zpage /* place following variables into
zero page */
#define BIT_SET(x,bitNo) ((x) |= 1<<(bitNo))
#define BIT_CLR(x,bitNo) ((x) &= ~(1<<(bitNo)))
char i, j;
struct {
    unsigned int b0:1;
    unsigned int b1:1;
    
```

```

} B;

void main(void) {
    /* demo using BSET/BCLR/BRSET/BRCLR */
    if (i&1) { /* BRCLR */
        /* BCLR: clearing a bit */
        i &= ~1; /* using normal ANSI-C */
        BIT_CLR(j,0); /* using a macro */
        __asm BCLR 0,i /* using inline assembly */
    }
    if ((i&1) == 0) { /* BRSET */
        /* BSET: setting a bit */
        i |= 1; /* using normal ANSI-C */
        BIT_SET(j,0); /* using a macro */
        __asm BCLR 0,j /* using inline assembly */
    }
    if (i&4) { /* BRCLR */
        i &= ~4; /* BCLR */
    }
    if ((i&4) == 4) { /* BRSET */
        i |= 4; /* BSET */
    }
    /* demo using bitfields in ANSI-C (warning: not portable, depends on
       the compiler
       how bitfields are allocated! */
    if (B.b0) {
        B.b1 = 1;
    } else if (B.b1) {
        B.b0 = 0;
    }
}
#endif
13: void main(void) {
14:     /* demo using BSET/BCLR/BRSET/BRCLR */
15:     if (i&1) { /* BRCLR */
16:         /* BCLR: clearing a bit */
00000000 010006          BCLR  0,i,*6          /abs = 0009
17:         i &= ~1; /* using normal ANSI-C */
00000003 1100          BCLR  0,i
18:         BIT_CLR(j,0); /* using a macro */
00000005 1100          BCLR  0,j
19:         __asm BCLR 0,i /* using inline assembly */
20:     }
00000007 1100          BCLR  0,i
21:     if ((i&1) == 0) { /* BRSET */
22:         /* BSET: setting a bit */
00000009 000006          BRSET 0,i,*6          /abs = 0012
23:         i |= 1; /* using normal ANSI-C */

```

## HC(S)08 Backend

### Data Types

---

```
0000000C 1000          BSET    0,i
 24:      BIT_SET(j,0); /* using a macro */
0000000E 1000          BSET    0,j
 25:      __asm BCLR 0,j /* using inline assembly */
 26:      }
00000010 1100          BCLR    0,j
 27:      if (i&4) { /* BRCLR */
00000012 050002          BRCLR   2,i,*2      /abs = 0017
 28:      i &= ~4; /* BCLR */
 29:      }
00000015 1500          BCLR    2,i
 30:      if ((i&4) == 4) { /* BRSET */
00000017 050002          BRCLR   2,i,*2      /abs = 001C
 31:      i |= 4; /* BSET */
 32:      }
 33:      /* demo using bitfields in ANSI-C (warning: not portable,
        depends on the compiler
        how bitfields are allocated! */
0000001A 1400          BSET    2,i
 35:      if (B.b0) {
0000001C 010003          BRCLR   0,B,*3      /abs = 0022
 36:      B.b1 = 1;
0000001F 1200          BSET    1,B
00000021 81          RTS
 37:      } else if (B.b1) {
00000022 0300FC          BRCLR   1,B,*-4     /abs = 0021
 38:      B.b0 = 0;
 39:      }
00000025 1100          BCLR    0,B
 40:      }
00000027 81          RTS
#endif
```

---

## Pointer Types and Function Pointers

Table 10.4 shows data pointer sizes depending on memory model and `__far` or `__near` keyword usage:

**Table 10.4 Data Pointer Sizes**

<b>Memory Model</b>	<b>Compiler Option</b>	<b>Default Pointer Size</b>	<b><u>near</u> Pointer Size</b>	<b><u>far</u> Pointer Size</b>
SMALL	-Ms	2	1	2
TINY	-Mt (default)	1	1	2

Function pointers have always the size of 2 bytes.

## Structured Types and Alignment

Objects allocated in memory are *not* aligned. Like objects, elements of an array and struct or union members are not aligned.

Local variables are allocated on the stack (which is growing downwards).

The most significant part of a simple variable is always stored at the low memory address (big endian).

---

**NOTE** The Compiler is free to align variables and fields. Always use implementation-independent access.

---

## Object Size

The maximum size of an object is 32 kilobytes.

## Register Usage

The Compiler uses all the standard registers of the MC68HC08. The stack pointer SP is used as stack pointer and as a frame pointer.

# Call Protocol and Calling Conventions

The calling protocols of the native HC08 and the new HCS08 are different. For the HCS08, the H register is loaded and stored more easily together with the X register. Thus, for the HCS08, the H register is used for parameter passing and return values.

## HC08 Argument Passing

The C calling convention is used for all functions. The caller pushes the arguments from left to right. After the call, the caller removes the parameters from the stack.

If a function with a fixed number of arguments and the size of the last parameter is two bytes, the last parameter is passed in X and A.

If a function with a fixed number of arguments and the size of the last parameter is one byte and the size of the second last parameter is more than one, the last parameter is passed in A. If the size of the second last parameter is also one byte, the second last parameter is passed in A and the last one in X.

## HCS08 Argument Passing (used for the -Cs08 option)

The C calling convention is used for all functions. The caller pushes the arguments from left to right. After the call, the caller removes the parameters from the stack.

If a function with a fixed number of arguments and the size of the last parameter is two bytes, the last parameter is passed in H and X.

If a function with a fixed number of arguments and the size of the last parameter is one byte, the last parameter is passed in A. If the size of the second last parameter is also one byte, the second last parameter is passed in X. If the size of the second last parameter is two bytes, the second last parameter is passed in H and X.

## HC08 Return Values

Function results are returned in registers, except if the function returns an object with a size greater than two bytes. Different registers are used depending on the return type, as shown in Table 10.5.

**Table 10.5 HC08 Return Values**

<b>Return Type</b>	<b>Registers</b>
char (signed or unsigned)	A
int (signed or unsigned)	X:A
pointers/arrays	X:A
function pointers	X:A

## HCS08 Return Values (used for the -Cs08 Option)

Function results are returned in registers, except if the function returns an object with a size greater than two bytes. Different registers depend on the return type, as shown in Table 10.6.

Table 10.6 HCS08 Return Values

Return Type	Registers
char (signed or unsigned)	A
int (signed or unsigned)	H:X
pointers or arrays	H:X
function pointers	H:X

## Returning Large Objects

Functions returning objects that are larger than two bytes are called with an additional parameter, which is passed in H:X. This parameter is the address to which the object should get copied.

## Stack Frames

A stack frame is a database used for passing parameters to and from a function using a stack in RAM.

## Frame Pointer

Functions normally have a stack frame containing all their local data. The Compiler does not set up an explicit frame pointer, but local data and parameters on the stack are accessed relative to the SP register.

## Entry Code

The normal *entry code* is a sequence of instructions reserving space for local variables (Listing 10.2):

## HC(S)08 Backend

### Call Protocol and Calling Conventions

---

#### Listing 10.2 Entry code

---

```
PSHA      ; Only if there are register parameters
PSHX      ; Only if there are register parameters
AIS #(-s) ; Reserved space for local variables and spills
```

---

where *s* is the size (in bytes) of the local data of the function. There is no “static link”, and the “dynamic link” is not stored explicitly.

## Exit Code

*Exit code* removes local variables from the stack and returns to the caller (Listing 10.3):

#### Listing 10.3 Exit code

---

```
AIS #(t) ; Remove local stack space,
          ; including an eventual register parameter
RTS      ; Return to caller
```

---

## Pragma TRAP\_PROC

This pragma defines an interrupt routine; i.e., a function with this pragma active is terminated by an RTI instruction instead of an RTS. Also, the H register is saved and restored at the entry response exit of the interrupt routine. If you are sure that H is not written in the interrupt routine, you can use the `#pragma TRAP_PROC: Mark function as interrupt Function` pragma to disable the saving and restoring of H.

## Interrupt Vector Table Allocation

The Compiler provides a non-ANSI compliant way to directly specify the interrupt vector number in the source:

```
void interrupt 0 ResetFunction(void) {
    /* reset handler */
}
```

The Compiler uses the translation from interrupt vector number to interrupt vector address shown in Table 10.7.



**Table 10.7 Interrupt Vector Translation To Vector Address**

Vector Number	Vector Address	Vector Address Size
0	0xFFFFE, 0xFFFFF	2
1	0xFFFFC, 0xFFFFD	2
2	0xFFFFA, 0xFFFFB	2
...	...	...
n	0xFFFFF - (n*2)	2

## Segmentation

The memory space of the Linker may be partitioned into several segments. The Compiler allows attributing a certain segment name to certain global variables, or functions which are then allocated into that segment by the Linker. Where that segment actually lies is determined by an entry in the Linker parameter file.

There are two pragmas that specify code segments and data segments:

```
#pragma CODE_SEG [ __NEAR_SEG | __FAR_SEG | __SHORT_SEG ]
<name>
#pragma DATA_SEG
[ __DPAGE_SEG | __PPAGE_PAGE | __EPAGE_SEG | __SHORT_SEG ] <n>
```

Both are valid until the next pragma of the same kind is encountered. If no segment is specified, the Compiler assumes two default segments named `DEFAULT_ROM` (the default code segment) and `DEFAULT_RAM` (the default data segment). To explicitly make these default segments the current ones, use the segment name `DEFAULT ( )`:

### Listing 10.4 Explicit default segments

---

```
#pragma CODE_SEG DEFAULT
#pragma DATA_SEG DEFAULT
```

---

The additional keyword `__SHORT_SEG` informs the Compiler that a data segment is allocated in the zero page (address range from `0x0000` to `0x00FF`):

```
#pragma DATA_SEG __SHORT_SEG <segment_name>
or
#pragma DATA_SEG __SHORT_SEG DEFAULT
```

---

Using the zero page enables the Compiler to generate much denser code because the DIRECT addressing mode is used instead of EXTENDED.

---

**NOTE** It is the programmer's responsibility to actually allocate `__SHORT_SEG` segments in the zero page in the Linker parameter file. For more information, see the Linker section of the Build Tools manual.

---

## Optimizations

The Compiler applies a variety of code improving techniques commonly defined as "optimization". This section gives a short overview about the most important optimizations.

### Lazy Instruction Selection

Lazy instruction selection is a very simple optimization that replaces certain instructions by shorter and/or faster equivalents. Examples are the use of `TSTA` instead of `CMP #0` or using `COMA` instead of `EORA #0xFF`.

### Strength Reduction

Strength reduction is an optimization that replaces expensive operations by cheaper ones, where the cost factor is either execution time or code size. Examples are the replacement of multiplications and divisions by constant powers of two with left or right shifts.

### Shift Optimizations

Shifting a byte variable by a constant number of bits is intensively analyzed. The Compiler always tries to implement such shifts in the most efficient way. As an example, consider the following:

```
char a, b;  
a = b << 4;
```

which disassembles in the following code:

```
LDA    b  
NSA                ; Swap nibbles...  
AND    #-16 ; ...and mask!  
STA    a
```

## Accessing Bitfields

Accesses to bitfields are inefficient due to the mask and shift operations involved in (un)packing a bitfield. However, accesses to bitfields only one bit wide often get translated well using the instructions `BSET/BCLR`, and the bit branches `BRSET` and `BRCLR`, if the addressing mode is direct.

To enable direct addressing, declare the bitfield in the zero page using the `DATA_SEG __SHORT_SEG` pragma (Listing 10.5).

### Listing 10.5 Example

---

```
#pragma DATA_SEG __SHORT_SEG DATA_ZEROPAGE;
struct {
    int a:1;
} bf;

void main(void) {
    bf.a = -1;
}
```

---

## HC08 Branch Optimizations

This optimization tries to minimize the span of branch instructions, or in other words: the Compiler replaces a relative branch by a branch with the inverted condition across an unconditional jump, if the offset of this branch is not in the range [-128 to 127]:

```
BRcc dest
...; More than 127 bytes of code
dest:
```

This is changed (provided the code indicated by “...” doesn’t contain another branch to label `dest`, so that it might be possible to have the `BRcc` jump to that branch) to:

```
BR~cc skip
JMP dest
skip:
... ; More than 127 bytes of code
dest:
```

Also, branches to branches may be resolved into two branches to the same target. Redundant branches (e.g., a branch to the instruction immediately following it) may be removed.

An unconditional branch over one byte is replaced by the opcode byte of `BRN` (branch never). The following byte is skipped (opcode decoded as `SKIP1`). An unconditional

branch over two bytes is replaced by the opcode of `CPHX <immediate_16>` if no flags are needed afterwards. The following two bytes are skipped (opcode decoded as `SKIP2`). One byte is gained for `SKIP1` and 2 for `SKIP2`. The execution speed remains unchanged. See the `-OnB: Disable Branch Optimizer` compiler option to disable this optimization.

## Optimization for Execution Time or Code Size

There are various points where the Compiler has to choose between two possibilities: it can either generate fast, but large code, or small but slower code.

The Compiler optimizes on code size. It often has to decide between a runtime routine and expanded code. In these cases, the runtime routine is chosen only if it is at least three bytes shorter than the expanded instruction sequence.

## Volatile Objects

The Compiler does not do register tracing on volatile global objects. Accesses to volatile global objects are not eliminated.

# Generating Compact Code with the CHC08 Compiler

Some compiler options or use of `__SHORT_SEG` segments help you to generate compact code with the CHC08 compiler.

## Compiler Options

Using the following compiler option helps to reduce the size of the code generated:

`-Cni`: Non integral promotion on integer

When this option is specified, the ANSI C integral promotion does not apply to character comparison or arithmetic operations. This saves a large amount of code.

## \_\_SHORT\_SEG Segments

Variables allocated on the direct page (between 0 and `0xFF`) are accessed using the direct addressing mode. The Compiler will allocate some variables on the direct page, if they are defined in a `__SHORT_SEG` segment.

**Listing 10.6 Example**

---

```
#pragma DATA_SEG __SHORT_SEG myShortSegment
unsigned int myVar1, myVar2;
#pragma DATA_SEG DEFAULT
unsigned int myvar3, myVar4.
```

---

In the previous example, ‘myVar1’ and ‘myVar2’ are both accessed using the direct addressing mode. The variables, ‘myVar3’ and ‘myVar4’, are accessed using the extended addressing mode.

When some exported variables are defined in a \_\_SHORT\_SEG segment, the external declaration for these variables must also specify that they are allocated in a \_\_SHORT\_SEG segment.

**Listing 10.7 The external definition of the variable defined above looks like:**

---

```
#pragma DATA_SEG __SHORT_SEG myShortSegment
extern unsigned int myVar1, myVar2;
#pragma DATA_SEG DEFAULT
extern unsigned int myvar3, myVar4.
```

---

The segment must be placed on the direct page in the PRM file.

**Listing 10.8 Example**

---

```
LINK test.abs
NAMES test.o start08.o ansi.lib END
SECTIONS
    Z_RAM = READ_WRITE 0x0080 TO 0x00FF;
    MY_RAM = READ_WRITE 0x0100 TO 0x01FF;
    MY_ROM = READ_ONLY 0xF000 TO 0xFEFF;
PLACEMENT
    DEFAULT_ROM INTO MY_ROM;
    DEFAULT_RAM INTO MY_RAM;
    _DATA_ZEROPAGE, myShortSegment INTO Z_RAM;
END
STACKSIZE 0x60
VECTOR ADDRESS 0xFFFE _Startup /* set reset vector on _Startup */
```

---

Be careful: The linker is case-sensitive. The segment name must be identical in the C and PRM files.

## HC(S)08 Backend

Generating Compact Code with the CHC08 Compiler

---

If all the data and stack fits into the zero page, for convenience, use the tiny memory model.

## Defining I/O Registers

The HC08 I/O Registers are usually based at address 0. In order to tell the compiler it should use direct addressing mode to access the I/O registers, these registers are defined in a `__SHORT_SEG` section based at the specified address.

In the C source file, the I/O registers are defined as follows:

### Listing 10.9 I/O Example

---

```
typedef struct {
    unsigned char SCC1;
    unsigned char SCC2;
    unsigned char SCC3;
    unsigned char SCS1;
    unsigned char SCS2;
    unsigned char SCD;
    unsigned char SCBR;
} SCIstruct;
#pragma DATA_SEG __SHORT_SEG SCIRegs
volatile SCIstruct SCI;
#pragma DATA_SEG DEFAULT
```

---

The segment must be placed at the appropriate address in the PRM file.

### Listing 10.10 Example PRM file

---

```
LINK test.abs
NAMES test.o start08.o ansi.lib END
SECTIONS
    SCI_RG = READ_WRITE 0x0013 TO 0x0019;
    Z_RAM = READ_WRITE 0x0080 TO 0x00FF;
    MY_RAM = READ_WRITE 0x0100 TO 0x01FF;
    MY_ROM = READ_ONLY 0xF000 TO 0xFEFF;
PLACEMENT
    DEFAULT_ROM INTO MY_ROM;
    DEFAULT_RAM INTO MY_RAM;
    _DATA_ZEROPAGE INTO Z_RAM;
    SCIRegs INTO SCI_RG;
END
STACKSIZE 0x60
```

---

```
VECTOR ADDRESS 0xFFFFE _Startup /* set reset vector on _Startup */
```

---

Be careful: The linker is case-sensitive. The segment name must be identical in the C and PRM files.

## **HC(S)08 Backend**

*Generating Compact Code with the CHC08 Compiler*

---



# High-Level Inline Assembler for the Freescale HC(S)08

The HLI (High-Level Inline) Assembler provides makes full use of the properties of the target processor from within a C program. There is no need to write a separate assembly file, assemble it, and later bind it with the rest of the application written in ANSI-C with the inline assembler. The Compiler does all the work. For detailed information, refer to the Freescale HC(S)08 Family Reference Manual.

## Syntax

Inline assembly statements can appear anywhere a C statement can appear (an `__asm` statement must be inside a C function). Inline assembly statements take one of two forms, which have various configurations. If you use the first form, multiple `__asm` statements are on one line and comments are delimited like regular C or C++ comments. If you use the second form, one to several assembly instructions are contained within the `__asm` block, but only one assembly instruction per line is possible. Also, the semicolon starts an assembly comment.

```
"__asm" <Assembly Instruction> ";" ["/*" Comment "*/"]
```

```
"__asm" <Assembly Instruction> ";" ["//" Comment]
```

or

```
"__asm" "{"
  { <Assembly Instruction> [";" Comment] "\n" }
}"
```

or

```
"__asm" " (" <Assembly Instruction> [";" Comment] ")" ";"
```

or

```
"__asm" [" (" ] <string Assembly instruction > [")"] [";"]
```

with <string Assembly instruction >

```
= <Assembly Instruction> [";" <Assembly instruction>]
```

```
or
"#asm"
<Assembly Instruction> [";" Comment] "\n"
"#endasm"
```

## C Macros

The C macros are expanded inside of inline assembler code as they are expanded in C. One special point to note is the syntax of the `__asm` directive (generated by macros). As macros always expand to one single line, only the first form of the `__asm` keyword

```
__asm NOP;
```

is used in macros.

e.g.,

```
#define SPACE_OK { __asm NOP; __asm NOP; }
```

Using the second form is illegal:

```
#define NOT_OK { __asm { \
                    NOP; \
                    NOP; \
                }
```

Here the `NOT_OK` macro is expanded by the preprocessor to one single line, which is mistranslated because every assembly instruction must be explicitly terminated by a newline.

To use `#` inside macros to build immediates, use the pragma `NO_STRING_CONSTR`.

## Inline Assembly Language

First, the EBNF Syntax of an inline assembly instruction is provided. A short discussion on semantics and special constructs follows.

```
AsmLine      = [Label](Code | Directive).
Label        = ident :.
Code         = mnemonic [ArgList][OptParams].
Directive    = Code.
ArgList      = Argument {, Argument}.
Argument     = Imm | Expression.
Imm          = # Expression.
OptParams    = ! RegList , RegList.
RegList      = { [Reg] {, Reg} }.
```

```
Reg           = A|HX|H|X|SR.
Variable      = Identifier.
Expression    = [Term {(+|-) Term}] [, (X|X+|HX|SP)].
Term          = Factor {(*/) Factor}.
Factor        = ( Expression )
              | - Factor
              | @ Factor
              | * Factor
              | Factor {: Factor|MSB)}
              | Number
              | TypeQualifier
              | Variable {. Field}
              | Procedure
              | Label
TypeQualifier = Type . Field {. Field}.
Procedure     = Identifier.
Label         = Identifier.
Type          = Identifier.
Field         = Identifier.
```

To resolve grammar ambiguities, the longest possible match is taken to reduce to Factor (shift is always preferred). The pseudo offset `MSB` designates the most significant byte of an effective address and is only allowed for memory operands. The `TypeQualifier` production above evaluates to the offset of the given field. `Type` must be the name of a struct type. All field given names must exist.

## Register Indirect Addressing Mode

The offset must be constant for the register indirect addressing modes. This constant may be an immediate, or a variable address that is relocated by the linker.

Example:

```
    STA    @Variable_Name,X
```

## Example

A simple example illustrates the use of the HLI-Assembler.

Assuming that `str` points to a character array, you can write a simple function in assembly language to determine the length of a string:

**Listing 11.1 strlen() definition**

---

```
int strlen (char *str)
  /*** The 'str' character array is passed on the stack. strlen returns
      length of 'str'.
      This procedure assumes len(str) is smaller than 256! */
{
  __asm {
    LDHX str          ; load pointer
    CLRA              ; init counter
    BRA test         ; go to test
  loop:
    AIX #1           ; increment pointer
    INCA             ; increment counter
  test:
    TST 0,X         ; not end of string?
    BNE loop        ; next char
    CLRX            ; return value in X:A(see later)
  };
  /* C statements could follow here */
}
```

---

---

**NOTE** If pragma NO\_ENTRY is not set, the Compiler takes care of entry and exit code. You do not have to worry about setting up a stack frame.

---

## Special Features

The remainder of this section covers special HLI features for the HC(S)08.

### Caller/Callee Saved Registers

Because the compiler does not save any registers on the caller/callee side, you do not have to save or restore any registers in the HLI over function calls.

### Reserved Words

The inline assembler knows some reserved words which must not collide with user-defined identifiers such as variable names. These reserved words are:

- all opcodes (LDA, STX, ...)
- all register names (A, H, X, HX, SR, SP)
- the identifier MSB

These reserved words are not case-sensitive in the inline assembler; i.e., LDA is the same as lda or even LdA. For all other identifiers (labels, variable names and so on), the inline assembler is case-sensitive.

## Pseudo-Opcodes

The inline assembler provides some pseudo opcodes to put constant bytes into the instruction stream. These are:

- DCB 1 ; Byte constant 1
- DCB 0 ; Byte constant 0
- DCW 12 ; Word constant 12
- DCL 20, 23 ; Longword constants
- DCF 1.85 ; IEEE 32bit float
- DCD 2.0 ; double IEEE64

## Accessing Variables

The inline assembler allows accessing local and global variables declared in C – simply use their names in the instruction.

## Address Notation

In addition to the address syntax described in the HCS08 Family Manual, one may access objects using a C-like syntax and symbolic variable names. In this case all addresses are assumed to be of type `char *`.

```
@var          ==> address of var
*var          ==> object pointed to by var
var:offs     ==> *((@var) + offs)
obj.field    ==> *((@obj) + offset_of(field))
type.field   ==> offset_of(field)
```

## H:X Instructions

LDHX is allowed with any reasonable argument and is not restricted to DIRECT page and IMMEDIATE arguments. In the native HC08, they represent code patterns which are automatically translated into an appropriate code sequence if the option `-cs08` is not active. Consider the following example in Table 11.1 - assume `local` is a variable on the stack:

Table 11.1 H:X Instructions

LDHX local	LDX local:0 PSHX PULH LDX local:1
LDHX @local	TSX AIX #<local-offset>
<b>Inline Assembly</b>	<b>Expansion</b>

## Constant Expressions

Constant expressions may be used anywhere an IMMEDIATE value is expected to contain the binary operators for addition ("+"), subtraction ("-"), multiplication ("\*"), and division ("/"). Also, the unary operator "-" is allowed. Round brackets may be used to force an evaluation order other than the normal one. The syntax of numbers is the same as in ANSI-C.

A constant expression may also be the address of a global variable or the offset of a local variable.

## Optimizing Inline Assembly

The Compiler is free to modify the instruction stream as long as its semantics remains the same. This is especially true for stack pointer indirect accesses (*ind*, *SP*, e.g., accesses to local variables), which may be replaced by *TSX* and a *ind*, *H:X* access.

## Assertions

The pseudo instruction

```
_ASSERT <reg>, addr
```

declares that the register *<reg>* (one of *H*, *X*, *HX*, *A*) contains *addr* at this instruction. It may be used by the Compiler to perform optimizations. The code will still be correct if this information is ignored. A typical example might be (Listing 11.2):

Listing 11.2 Using `_ASSERT`

---

```
LDHX @a:4
MOV #4, tmp
loop:
AIX #-1
LDA 0,X
```

```
STA    4,X
DBNZ  tmp, loop
_ASSERT HX, @a
```

---

## Stack Adjust

The `_STACK` pseudo instruction is not supported any longer with the ICG-based version of the compiler.

The pseudo instruction:

```
label:
    _STACK #offs
```

declares that the stack pointer's value at this instruction is (stack pointer's value at label) + offs. It is ignored by the ICG-based Compiler.

The pseudo instruction:

```
_ADJ #offs
label:
```

adjusts the SP register with appropriate AIS instructions, so that `_STACK` holds (value at label+offs) afterwards. The `_STACK` instruction is useful for situations like the following (Listing 11.3):

### Listing 11.3 Using the `_STACK` pseudo instruction

---

```
base:
    CLRA
    MOV  #8, tmp
loop:
    PSHA
    DBNZ tmp, loop
    _STACK #-8      ; The pseudo instruction
_STACK label:offs
```

---

## In & Gen Sets

The pseudo instruction

```
_INGEN <in-set>, <gen-set>
```

## High-Level Inline Assembler for the Freescale HC(S)08

### Special Features

---

announces to the Compiler that registers in <in-set> are used further, and those in <gen-set> are not. The notation:

```
<instr> <args> ! <in-set>, <gen-set>
```

declares which registers are used and which ones are modified by an instruction. This works well for JSR, JMP, and RTS (and similar) instructions. The default is always on the conservative side. Consider the following example (Listing 11.4):

#### Listing 11.4 Example program

---

```
LDA a:1
ADD b:1
STA b:1
LDA a:0
ADC b:0
STA b:0
RTS ! {}, {}
```

---

This allows the Compiler to discard the H:X register for accesses to local variables. With the simple RTS, the Compiler does not know if the H:X register is to be preserved or not. Multiple registers must be separated by a '+'. For example:

```
RTS ! {A+X}, {}
```

## Getting the High Address Part in HLI

It is possible to get the high address part of an object using the MSB syntax. The following example illustrates this (Listing 11.5):

#### Listing 11.5 Getting the high-address part of an object

---

```
int i, *p_i;

void main(void) {
    /* p_i = &i */
    __asm {
        LDA @i
        STA p_i:1
        LDA @i:MSB
        STA p_i
    }
}
```

---



The following examples illustrate the usage with the original code and the generated one:  
(Listing 11.6).

**Listing 11.6 Four examples of HLI usage and their generated assembly code**

---

```

char f(void);                char f(void);

char asm1(int a) {          char asm1(int a) {
    int b;                  int b;
    __asm {                __asm {
        LDA    a:1          LDA 5,X ; H:X is used
        STA    b:1          STA 1,X ; to access all locals
        JSR    f ! {}, {}   JSR f
        LDA    b:1          STA 4,X
        RTS    ! {}, {A}    LDA 1,X
    }                       RTS
}                             }

char asm2(int a) {          char asm2(int a) {
    int b;                  int b;
    __asm {                __asm {
        LDA    a:1          LDA 6,SP ; X is input register
        STA    b:1          STA 2,SP ; for f. SP must be used
        JSR    f ! {X}, {}  JSR f
        STA    a:0          TSX ; f does not set X.
        LDA    b:1          STA 4,X ; now X is free
        RTS    ! {}, {A}    LDA 1,X
    }                       RTS
}                             }

char asm3(int a) {          char asm3(int a) {
    int b;                  int b;
    __asm {                __asm {
        LDA    a:1          TSX
        STA    b:1          LDA 5,X ; H:X is used
        JSR    f ! {}, {X}  STA 1,X ; to access locals
        STA    a:0          JSR f ; f destroys X
        LDA    b:1          TSX ; X is restored
        RTS    ! {}, {A}    STA 4,X ; to access locals
    }                       LDA 1,X
}                             RTS
}                             }

char asm4(int a) {          char asm4(int a) {

```

## High-Level Inline Assembler for the Freescale HC(S)08

### Special Features

---

```
int b;                                int b;
__asm {                               __asm {
  LDA  a:1                             LDA  6,SP
  STA  b:1                             STA  2,SP
  JSR  f ! {}, {}                     JSR  f
  STA  a:0                             STA  5,SP
  LDA  b:1                             LDA  2,SP
  RTS  ! {X}, {A}                     RTS  ; X must not be used
                                        ; in this function!
}                                       }
}
```

---



# ANSI-C Library Reference

---

This section covers the ANSI-C Library.

- **Library Files:** Description of the types of library files
- **Special Features:** Description of special considerations of the ANSI-C standard library relating to embedded systems programming
- **Library Structure:** Examination of the various elements of the ANSI-C library, grouped by category.
- **Types and Macros in the Standard Library:** Discussion of all types and macros defined in the ANSI-C standard library.
- **The Standard Functions:** Description of all functions in the ANSI-C library



# Library Files

## Directory Structure

The library files are delivered in the following structure (:Listing 12.1).

**Listing 12.1** Layout of files after a CodeWarrior installation/

---

```

<install>\lib\<target>c\          /* readme files, make files */
<install>\lib\<target>c\src      /* C library source files */
<install>\lib\<target>c\include /* library include files */
<install>\lib\<target>c\lib     /* default library files */
<install>\lib\<target>c\prm     /* Linker parameter files */

```

---

Check out the README.TXT located in the library folder with additional information on memory models and library filenames.

## How to Generate a Library

In the directory structure above, a CodeWarrior \*.mcp file is provided to build all the libraries and the startup code object files. Simply load the <target>\_lib.mcp file into CodeWarrior and build all the targets.

## Common Source Files

Table 12.1 lists the source and header files of the Standard ANSI Library that are not target-dependent.

**Table 12.1** Standard ANSI Library—Target Independent Source and Header Files

Source File	Header File
alloc.c	
assert.c	assert.h

## Library Files

### Startup Files

**Table 12.1 Standard ANSI Library—Target Independent Source and Header Files**

Source File	Header File
ctype.c	ctype.h
	errno.h
heap.c	heap.h
	limits.h
math.c, mathf.c	limits.h, ieeemath.h, float.h
printf.c, scanf.c	stdio.h
signal.c	signal.h
	stdarg.h
	stddef.h
stdlib.c	stdlib.h
string.c	string.h
	time.h

## Startup Files

Because every memory model needs special startup initialization, there are also startup object files compiled with different Compiler option settings (see Compiler options for details).

The correct startup file has to be linked with the application depending on the memory model chosen. The floating point format used does not matter for the startup code.

Note that the library files contain a generic startup written in C as an example of doing all the tasks needed for a startup:

- Zero Out
- Copy Down
- Register initialization
- Handling ROM libraries

Because not all of the above tasks may be needed for an application and for efficiency reasons, special startup is provided as well (e.g., written in HLI). However, the version

---

written in C could be used as well. For example, just compile the ‘startup.c’ file with the memory/options settings and link it to the application.

## Startup Files for Freescale HC08

Depending on the memory model, a different startup object file has to be linked to the application: Table 12.2 lists the startup files for the HC08:

**Table 12.2 Startup Files for Motorola HC08**

<b>Startup Object File</b>	<b>Startup Source File</b>	<b>Compiler Options</b>
start08.o	Start08.c	-Ms
start08p.o	Start08.c	-Ms -C++f
start08t.o	Start08.c	-Mt
start08tp.o	Start08.c	-Mt -C++f

## Startup Files for Freescale HCS08

Depending on the memory model, a different startup object file has to be linked to the application. Table 12.3 lists the startup files for the HCS08:

**Table 12.3 Startup Files for Motorola HCS08**

<b>Startup Object File</b>	<b>Startup Source File</b>	<b>Compiler Options</b>
start08s.o	Start08.c	-Ms -Cs08
start08ts.o	Start08.c	-Mt -Cs08

## Library Files

Most of the object files of the ANSI library are delivered in the form of an object library (see below).

Several Library files are bundled with the Compiler. The reasons for having different library files are due to different memory models or floating point formats.

## Library Files

### *Library Files*

---

The library files contain all necessary runtime functions used by the compiler and the ANSI Standard Library as well. The list files (\* .lst extension) contains a summary of all objects in the library file.

To link against a modified file which also exists in the library, it must be specified first in the link order.

Please check out the readme.txt located in the library structure (lib<target>c\README.TXT) for a list of all delivered library files and memory model or options used.



# Special Features

---

Not everything defined in the ANSI standard library makes sense in embedded systems programming. Therefore, not all functions have been implemented, and some have been left open to be implemented because they strongly depend on the actual setup of the target system.

This chapter describes and explains these points.

---

**NOTE** All functions not implemented do a `HALT` when called. All functions are re-entrant, except `rand()` and `srand()` because these use a global variable to store the seed, which might give problems with light-weight processes. Another function using a global variable is `strtok()`, because it has been defined that way in the ANSI standard.

---

## Memory Management -- `malloc()`, `free()`, `calloc()`, `realloc()`; `alloc.c`, and `heap.c`

File `'alloc.c'` provides a full implementation of these functions. The only problems remaining are the question of where to put the heap, how big should it be, and what should happen when the heap memory runs out.

All these points can be solved in the `"heap.c"` file. The heap simply is viewed as a large array, and there is a default error handling function. Feel free to modify this function or the size of the heap to suit the needs of the application. The size of the heap is defined in `libdefs.h`, `LIBDEF_HEAPSIZE`.

## Signals - `signal.c`

Signals have been implemented in a very rudimentary way - as traps. This means, function `signal()` allows you to set a vector to some function of your own (which of course should be a `TRAP_PROC`), while function `raise()` is not implemented. If you decide to ignore a certain signal, a default handler doing nothing is installed.

## Special Features

*Multi-byte Characters - mblen(), mbtowc(), wctomb(), mbstowcs(), wcstombs(); stdlib.c*

---

# Multi-byte Characters - mblen(), mbtowc(), wctomb(), mbstowcs(), wcstombs(); stdlib.c

Because the compiler does not support multi-byte characters, all routines in "stdlib.c" dealing with those have not been implemented. If these functions are needed, the programmer will have to specifically write them.

## Program Termination - abort(), exit(), atexit(); stdlib.c

Because programs in embedded systems usually are not expected to terminate, we only provide a minimum implementation of the first two functions, while atexit() is not implemented at all. Both abort() and exit() simply perform a HALT.

## I/O - printf.c

The printf() library function is not implemented in the current version of the library sets in the ANSI libraries, but it is found in the "terminal.c" file.

This difference has been planned because often no terminal is available at all or a terminal depends highly on the user hardware.

The ANSI library contains several functions which makes it simple to implement the printf() function with all its special cases in a few lines.

The first, ANSI-compliant way is to allocate a buffer and then use the vsprintf() ANSI function (Listing 13.1).

### Listing 13.1 An implementation of the printf() function

---

```
int printf(const char *format, ...) {
    char outbuf[MAXLINE];
    int i;
    va_list args;
    va_start(args, format);
    i = vsprintf(outbuf, format, args);
    va_end(args);
    WriteString(outbuf);
    return i;
}
```

---

The value of `MAXLINE` defines the maximum size of any value of `printf()`. The `WriteString()` function is assumed to write one string to a terminal. There are several disadvantages of this solution:

- A buffer is needed which alone may use a large amount of RAM.
- As unimportant how large the buffer (`MAXLINE`) is, it is always possible that a buffer overflow occurs. Therefore this solution is not safe.

Two non-ANSI functions - `vprintf()` and `set_printf()` - are provided in its newer library versions in order to avoid both disadvantages.

Because these functions are a non-ANSI extension, they are not contained in the "stdio.h" header file.

Therefore, their prototypes must be specified before they are used (Listing 13.2):

---

### Listing 13.2 Prototypes of `vprintf()` and `set_printf()`

---

```
int vprintf(const char *pformat, va_list args);
void set_printf(void (*f)(char));
```

---

The `set_printf()` function installs a callback function, which is called later for every character which should be printed by `vprintf()`.

Be advised that the standard ANSI C `printf()` derivatives functions `sprintf()` and `vsprintf()` are also implemented by calls to `set_printf()` and `vprintf()`. This way much of the code for all `printf` derivatives can be shared across them.

There is also a limitation of the current implementation of `printf()`. Because the callback function is not passed as an argument to `vprintf()`, but held in a global variable, all the `printf()` derivatives are not reentrant. Even calls to different derivatives at the same time are not allowed.

For example, a simple implementation of a `printf()` with `vprintf()` and `set_printf()` is shown in Listing 13.3:

---

### Listing 13.3 Implementation of `printf()` with `vprintf()` and `set_printf()`

---

```
int printf(const char *format, ...){
    int i;
    va_list args;

    set_printf(PutChar);
    va_start(args, format);
    i = vprintf(format, args);
    va_end(args);
}
```

## Special Features

*Locales - locale.\**

---

```
    return i;  
}
```

---

The `PutChar()` function is assumed to print one character to the terminal.

Another remark has to be made about the `printf()` and `scanf()` functions. The full source code is provided of all `printf()` derivatives in "printf.c" and of `scanf()` in "scanf.c". Usually many of the features of `printf()` and `scanf()` are not used by a specific application. The source code of the library modules `printf` and `scanf` contains switches (defines) to allow the use to switch off unused parts of the code. This especially includes the large floating-point parts of `vprintf()` and `vscanf()`.

## Locales - locale.\*

Has not been implemented.

## ctype

`ctype` contains two sets of implementations for all functions. The standard is a set of macros which translate into accesses to a lookup table.

This table uses 257 bytes of memory, so an implementation using real functions is provided. These are accessible if the macros are undefined first. After `#undef isupper`, `isupper` is translated into a call to function `isupper()`. Without the `"undef"`, `"isupper"` is replaced by the corresponding macro.

Using the functions instead of the macros of course saves RAM and code size - at the expense of some additional function call overhead.

## String Conversions - `strtol()`, `strtoul()`, `strtod()`, and `stdlib.c`

To follow the ANSI requirements for string conversions, range checking has to be done. The variable "errno" is set accordingly and special limit values are returned. The macro "ENABLE\_OVERFLOW\_CHECK" is set to 1 by default. To reduce code size it is recommended to switch off this macro (set `ENABLE_OVERFLOW_CHECK` to 0).

# Library Structure

---

In this section, the various parts of the ANSI-C standard library are examined, grouped by category. This library not only contains a rich set of functions, but also numerous types and macros.

## Error Handling

Error handling in the ANSI library is done using a global variable `errno` that is set by the library routines and may be tested by a user program. There also are a few functions for error handling:

```
void  assert(int expr);
void  perror(const char *msg);
char * strerror(int errno);
```

## String Handling Functions

Strings in ANSI-C always are null-terminated character sequences. The ANSI library provides the following functions to manipulate such strings (Listing 14.1).

### Listing 14.1 ANSI-C String Manipulation Functions

---

```
size_t strlen(const char *s);
char * strcpy(char *to, const char *from);
char * strncpy(char *to, const char *from, size_t size);
char * strcat(char *to, const char *from);
char * strncat(char *to, const char *from, size_t size);
int   strcmp(const char *p, const char *q);
int   strncmp(const char *p, const char *q, size_t size);
char * strchr(const char *s, int ch);
char * strrchr(const char *s, int ch);
char * strstr(const char *p, const char *q);
size_t strspn(const char *s, const char *set);
size_t strcspn(const char *s, const char *set);
char * strpbrk(const char *s, const char *set);
```

## Library Structure

### *Memory Block Functions*

---

```
char * strtok(char *s, const char *delim);
```

---

## Memory Block Functions

Closely related to the string handling functions are those operating on memory blocks. The main difference to the string functions is that they operate on any block of memory, whether it is null-terminated or not. The length of the block must be given as an additional parameter. Also, these functions work with `void` pointers instead of `char` pointers (Listing 14.2).

### Listing 14.2 ANSI-C Memory Block Functions

---

```
void * memcpy(void *to, const void *from, size_t size);  
void * memmove(void *to, const void *from, size_t size);  
int memcmp(const void *p, const void *q, size_t size);  
void * memchr(const void *adr, int byte, size_t size);  
void * memset(void *adr, int byte, size_t size);
```

---

## Mathematical Functions

The ANSI library contains a variety of floating point functions. The standard interface, which is defined for type `double` (Listing 14.3), has been augmented by an alternate interface (and implementation) using type `float` (Listing 14.4).

### Listing 14.3 ANSI-C Double-Precision Mathematical Functions

---

```
double acos(double x);  
double asin(double x);  
double atan(double x);  
double atan2(double x, double y);  
double ceil(double x);  
double cos(double x);  
double cosh(double x);  
double exp(double x);  
double fabs(double x);  
double floor(double x);  
double fmod(double x, double y);  
double frexp(double x, int *exp);  
double ldexp(double x, int exp);  
double log(double x);
```

---

```
double log10(double x);
double modf(double x, double *ip);
double pow(double x, double y);
double sin(double x);
double sinh(double x);
double sqrt(double x);
double tan(double x);
double tanh(double x);
```

---

The functions using the `float` type have the same names with an "f" appended.

---

**Listing 14.4 ANSI-C Single-Precision Mathematical Functions**

---

```
float acosf(float x);
float asinf(float x);
float atanf(float x);
float atan2f(float x, float y);
float ceilf(float x);
float cosf(float x);
float coshf(float x);
float expf(float x);
float fabsf(float x);
float floorf(float x);
float fmodf(float x, float y);
float frexpf(float x, int *exp);
float ldexpf(float x, int exp);
float logf(float x);
float log10f(float x);
float modff(float x, float *ip);
float powf(float x, float y);
float sinf(float x);
float sinhf(float x);
float sqrtf(float x);
float tanf(float x);
float tanhf(float x);
```

---

In addition, the ANSI library also defines a couple of functions operating on integral values (Listing 14.5):

---

**Listing 14.5 ANSI functions with integer arguments**

---

```
int    abs(int i);
div_t  div(int a, int b);
long   labs(long l);
```

---

## Library Structure

### Memory Management

---

```
ldiv_t ldiv(long a, long b);
```

Furthermore, the ANSI-C library contains a simple pseudo random number generator (Listing 14.6) and a function using a seed to start the random-number generator:

#### Listing 14.6 Random-number generator functions.

---

```
int rand(void);
void srand(unsigned int seed);
```

---

## Memory Management

To allocate and deallocate memory blocks, the ANSI library provides the following functions:

- `void* malloc(size_t size);`
- `void* calloc(size_t n, size_t size);`
- `void* realloc(void* ptr, size_t size);`
- `void free(void* ptr);`

Because it is not possible to implement these functions in a way that suits all possible target processors and memory configurations, all these functions are based on the system module `heap.c` file, which can be modified by the user to fit a particular memory layout.

## Searching and Sorting

The ANSI library contains both a generalized searching and a generalized sorting procedure:

---

```
void* bsearch(const void *key, const void *array,
              size_t n, size_t size, cmp_func f);

void qsort(void *array, size_t n, size_t size, cmp_func f);
```

---



## Character Functions

These functions test or convert characters. All these functions are implemented both as macros and as functions, and, by default, the macros are active. To use the corresponding function, you have to `#undef` the macro.

### Listing 14.7 ANSI-C Character Functions

---

```
int isalnum(int ch);
int isalpha(int ch);
int iscntrl(int ch);
int isdigit(int ch);
int isgraph(int ch);
int islower(int ch);
int isprint(int ch);
int ispunct(int ch);
int isspace(int ch);
int isupper(int ch);
int isxdigit(int ch);
int tolower(int ch);
int toupper(int ch);
```

---

The ANSI library also defines an interface for multibyte and wide characters. The implementation only offers minimum support for this feature: the maximum length of a multibyte character is one byte.

```
int    mblen(char *mbs, size_t n);
size_t mbstowcs(wchar_t *wcs, const char *mbs, size_t n);
int    mbtowc(wchar_t *wc, const char *mbc, size_t n);
size_t wcstombs(char *mbs, const wchar_t *wcs size_t n);
int    wctomb(char *mbc, wchar_t wc);
```

## System Functions

The ANSI standard includes some system functions for raising and responding to signals, non-local jumping, and so on.

### Listing 14.8 ANSI-C System Functions

---

```
void    abort(void);
int     atexit(void(* func) (void));
void    exit(int status);
```

---

## Library Structure

### *Time Functions*

---

```
char*      getenv(const char* name);
int        system(const char* cmd);
int        setjmp(jmp_buf env);
void       longjmp(jmp_buf env, int val);
_sig_func  signal(int sig, _sig_func handler);
int        raise(int sig);
```

---

To process variable length argument lists, the ANSI library provides the following “functions” (they are implemented as macros):

```
void va_start(va_list args, param);
type va_arg(va_list args, type);
void va_end(va_list args);
```

## Time Functions

In the ANSI library, there also are several function to get the current time. In an embedded systems environment, implementations for these functions cannot be provided because different targets may use different ways to count the time.

### **Listing 14.9 ANSI-C Time Functions**

---

```
clock_t     clock(void);
time_t      time(time_t *time_val);
struct tm * localtime(const time_t *time_val);
time_t      mktime(struct tm *time_rec);
char        * asctime(const struct tm *time_rec);
char        ctime(const time_t *time_val);
size_t      strftime(char *s, size_t n,
                    const char *format,
                    const struct tm *time_rec);
double      difftime(time_t t1, time_t t2);
struct tm * gmtime(const time_t *time_val);
```

---

## Locale Functions

These functions are for handling locales. The ANSI-C library only supports the minimal “C” environment (Listing 14.10).

**Listing 14.10 ANSI-C locale functions**

---

```
struct lconv *localeconv(void);
char          *setlocale(int cat, const char *locale);
int           strcoll(const char *p, const char *q);
size_t        strxfrm(const char *p, const char *q, size_t n);
```

---

## Conversion Functions

Functions for converting strings to numbers are found in Listing 14.11.

**Listing 14.11 ANSI-C string/number conversion functions**

---

```
int           atoi(const char *s);
long          atol(const char *s);
double        atof(const char *s);
long          strtol(const char *s, char **end, int base);
unsigned long strtoul(const char *s, char **end, int base);
double        strtod(const char *s, char **end);
```

---

## printf() and scanf()

More conversions are possible for the C functions for reading and writing formatted data. These functions are shown in Listing 14.12.

**Listing 14.12 ANSI-C read and write functions**

---

```
int sprintf(char *s, const char *format, ...);
int vsprintf(char *s, const char *format, va_list args);
int sscanf(const char *s, const char *format, ...);
```

---

## File I/O

The ANSI-C library contains a fairly large interface for file I/O. In microcontroller applications however, one usually does not need file I/O. In the few cases where one would need it, the implementation depends on the actual setup of the target system.

## Library Structure

### File I/O

---

Therefore, is therefore impossible for Freescale to provide an implementation for these features that the user has to specifically implement.

Listing 14.13 contains file I/O functions while Listing 14.14 has functions for the reading and writing of characters. The functions for reading and writing blocks of data are found in Listing 14.15. Functions for formatted I/O on files are found in Listing 14.16 and Listing 14.17 has functions for positioning data within files.

#### Listing 14.13 ANSI-C file I/O functions.

---

```
FILE* fopen(const char *name, const char *mode);
FILE* freopen(const char *name, const char *mode, FILE *f);
int  fflush(FILE *f);
int  fclose(FILE *f);
int  feof(FILE *f);
int  ferror(FILE *f);
void clearerr(FILE *f);
int  remove(const char *name);
int  rename(const char *old, const char *new);
FILE* tmpfile(void);
char* tmpnam(char *name);
void setbuf(FILE *f, char *buf);
int  setvbuf(FILE *f, char *buf, int mode, size_t size);
```

---

#### Listing 14.14 ANSI-C functions for writing and reading characters

---

```
int  fgetc(FILE *f);
char* fgets(char *s, int n, FILE *f);
int  fputc(int c, FILE *f);
int  fputs(const char *s, FILE *f);
int  getc(FILE *f);
int  getchar(void);
char* gets(char *s);
int  putc(int c, FILE *f);
int  puts(const char *s);
int  ungetc(int c, FILE *f);
```

---

#### Listing 14.15 ANSI-C functions for reading and writing blocks of data.

---

```
size_t fread(void *buf, size_t size, size_t n, FILE *f);
size_t fwrite(void *buf, size_t size, size_t n, FILE *f);
```

---

**Listing 14.16 ANSI-C formatted I/O functions on files.**

---

```
int fprintf(FILE *f, const char *format, ...);
int vfprintf(FILE *f, const char *format, va_list args);
int fscanf(FILE *f, const char *format, ...);
int printf(const char *format, ...);
int vprintf(const char *format, va_list args);
int scanf(const char *format, ...);
```

---

**Listing 14.17 ANSI-C positioning functions.**

---

```
int fgetpos(FILE *f, fpos_t *pos);
int fsetpos(FILE *f, const fpos_t *pos);
int fseek(FILE *f, long offset, int mode);
long ftell(FILE *f);
void rewind(FILE *f);
```

---

## **Library Structure**

*File I/O*

---

# Types and Macros in the Standard Library

This section discusses all types and macros defined in the ANSI standard library. We cover each of the header files, in alphabetical order.

## errno.h

This header file just declares two constants that are used as error indicators in the global variable `errno`.

```
extern int errno;
#define EDOM    -1
#define ERANGE -2
```

## float.h

Defines constants describing the properties of floating-point arithmetic. See Table 15.1 and Table 15.2.

**Table 15.1 Rounding and radix constants**

Constant	Description
FLT_ROUNDS	Gives the rounding mode implemented
FLT_RADIX	The base of the exponent

All other constants are prefixed by either `FLT_`, `DBL_` or `LDBL_`. `FLT_` is a constant for type `float`, `DBL_` for `double`, and `LDBL_` for `long double`.

## Types and Macros in the Standard Library

*limits.h*

---

**Table 15.2 Other constants defined in float.h**

Constant	Description
DIG	Number of significant digits.
EPSILON	Smallest positive $x$ for which $1.0 + x \neq x$ .
MANT_DIG	Number of binary mantissa digits.
MAX	Largest normalized finite value.
MAX_EXP	Maximum exponent such that $\text{FLT\_RADIX}^{\text{MAX\_EXP}}$ is a finite normalized value.
MAX_10_EXP	Maximum exponent such that $10^{\text{MAX\_10\_EXP}}$ is a finite normalized value.
MIN	Smallest positive normalized value.
MIN_EXP	Smallest negative exponent such that $\text{FLT\_RADIX}^{\text{MIN\_EXP}}$ is a normalized value.
MIN_10_EXP	Smallest negative exponent such that $10^{\text{MIN\_10\_EXP}}$ is a normalized value.

## limits.h

Defines a couple of constants for the maximum and minimum values that are allowed for certain types. See Table 15.3.

**Table 15.3 Constants defined in limits.h**

Constant	Description
CHAR_BIT	Number of bits in a character
SCHAR_MIN	Minimum value for signed char
SCHAR_MAX	Maximum value for signed char
UCHAR_MAX	Maximum value for unsigned char
CHAR_MIN	Minimum value for char
CHAR_MAX	Maximum value for char



**Table 15.3 Constants defined in limits.h (continued)**

Constant	Description
MB_LEN_MAX	Maximum number of bytes for a multi-byte character.
SHRT_MIN	Minimum value for short int
SHRT_MAX	Maximum value for short int
USHRT_MAX	Maximum value for unsigned short int
INT_MIN	Minimum value for int
INT_MAX	Maximum value for int
UINT_MAX	Maximum value for unsigned int
LONG_MIN	Minimum value for long int
LONG_MAX	Maximum value for long int
ULONG_MAX	Maximum value for unsigned long int

## locale.h

The header file in Listing 15.1 defines a `struct` containing all the locale-specific values.

**Listing 15.1 Locale-specific values**

```

struct lconv {
    /* "C" locale (default) */
    char *decimal_point; /* "." */
    /* Decimal point character to use for non-monetary numbers */
    char *thousands_sep; /* "" */
    /* Character to use to separate digit groups in
    the integral part of a non-monetary number. */
    char *grouping; /* "\CHAR_MAX" */
    /* Number of digits that form a group. CHAR_MAX
    means "no grouping", '\0' means take previous
    value.
    e.g., the string "\3\0" specifies the repeated
    use of groups of three digits. */
    char *int_curr_symbol; /* "" */
    /* 4-character string for the international
    currency symbol according to ISO 4217. The
    last character is the separator
    between currency symbol and amount. */
    char *currency_symbol; /* "" */

```

## Types and Macros in the Standard Library

*locale.h*

---

```
/* National currency symbol. */
char *mon_decimal_point; /* "." */
char *mon_thousands_sep; /* "" */
char *mon_grouping;      /* "\CHAR_MAX" */
/* Same as decimal_point etc., but for monetary numbers. */
char *positive_sign;     /* "" */
/* String to use for positive monetary numbers.*/
char *negative_sign;     /* "" */
/* String to use for negative monetary numbers. */
char int_frac_digits;    /* CHAR_MAX */
/* Number of fractional digits to print in a
   monetary number according to international format. */
char frac_digits;       /* CHAR_MAX */
/* The same for national format. */
char p_cs_precedes;     /* 1 */
/* 1 indicates that the currency symbol is left
   of a positive monetary amount; 0 indicates it is on the right. */
char p_sep_by_space;    /* 1 */
/* 1 indicates that the currency symbol is
   separated from the number by a space for
   positive monetary amounts. */
char n_cs_precedes;     /* 1 */
char n_sep_by_space;    /* 1 */
/* The same for negative monetary amounts. */
char p_sign_posn;       /* 4 */
char n_sign_posn;       /* 4 */
/* Defines the position of the sign for positive
   and negative monetary numbers:
   0 amount and currency are in parentheses
   1 sign comes before amount and currency
   2 sign comes after the amount
   3 sign comes immediately before the currency
   4 sign comes immediately after the currency */
};
```

---

There also are several constants that can be used in `setlocale()` to define which part of the locale should be set. See Table 15.4.

**Table 15.4 Constants used with `setlocale()`**

Constant	Description
LC_ALL	Changes the complete locale.
LC_COLLATE	Only changes the locale for functions <code>strcoll()</code> and <code>strxfrm()</code> .
LC_MONETARY	Changes the locale for formatting monetary numbers.

---

**Table 15.4 Constants used with `setlocale()` (*continued*)**

Constant	Description
LC_NUMERIC	Changes the locale for numeric, i.e., non-monetary formatting.
LC_TIME	Changes the locale for function <code>strftime()</code> .
LC_TYPE	Changes the locale for character handling and multi-byte character functions.

This implementation only supports the minimum “C” locale.

## math.h

Defines just this constant:

`HUGE_VAL`

Large value that is returned if overflow occurs.

## setjmp.h

Contains just this type definition:

```
typedef jmp_buf;
```

A buffer for `setjmp()` to store the current program state.

## signal.h

Defines signal-handling constants and types. See Table 15.5 and Table 15.6.

```
typedef sig_atomic_t;
```

## Types and Macros in the Standard Library

*stddef.h*

---

**Table 15.5 Constants defined in `signal.h`**

Constant	Definition
SIG_DFL	If passed as the second argument to <code>signal</code> , the default response is installed.
SIG_ERR	Return value of <code>signal()</code> , if the handler could not be installed.
SIG_IGN	If passed as the second argument to <code>signal()</code> , the signal is ignored.

**Table 15.6 Signal-t type macros**

Constant	Definition
SIGABRT	Abort program abnormally
SIGFPE	Floating point error
SIGILL	Illegal instruction
SIGINT	Interrupt
SIGSEGV	Segmentation violation
SIGTERM	Terminate program normally

## **stddef.h**

Defines a few generally useful types and constants. See Table 15.7.

**Table 15.7 Constants defined in `stddef.h`**

Constant	Description
<code>ptrdiff_t</code>	The result type of the subtraction of two pointers.
<code>size_t</code>	Unsigned type for the result of <code>sizeof</code> .
<code>wchar_t</code>	Integral type for wide characters.

**Table 15.7 Constants defined in `stddef.h` (*continued*)**

Constant	Description
<code>#define NULL ((void *) 0)</code>	
<code>size_t offsetof (type, struct_member)</code>	Returns the offset of field <code>struct_member</code> in <code>struct</code> type.

## stdio.h

There are two type declarations in this header file. See Table 15.8.

**Table 15.8 Type definitions in `stdio.h`**

Type Definition	Description
<code>FILE</code>	Defines a type for a file descriptor.
<code>fpos_t</code>	A type to hold the position in the file as needed by <code>fgetpos()</code> and <code>fsetpos()</code> .

Table 15.9 lists the constants defined in `stdio.h`.

**Table 15.9 Constants defined in `stdio.h`**

Constant	Description
<code>BUFSIZ</code>	Buffer size for <code>setbuf()</code> .
<code>EOF</code>	Negative constant to indicate end-of-file.
<code>FILENAME_MAX</code>	Maximum length of a filename.
<code>FOPEN_MAX</code>	Maximum number of open files.
<code>_IOFBF</code>	To set full buffering in <code>setvbuf()</code> .
<code>_IOLBF</code>	To set line buffering in <code>setvbuf()</code> .
<code>_IONBF</code>	To switch off buffering in <code>setvbuf()</code> .
<code>SEEK_CUR</code>	<code>fseek()</code> positions relative from current position.
<code>SEEK_END</code>	<code>fseek()</code> positions from the end of the file.

## Types and Macros in the Standard Library

*stdlib.h*

**Table 15.9** Constants defined in *stdio.h* (*continued*)

Constant	Description
SEEK_SET	fseek() positions from the start of the file.
TMP_MAX	Maximum number of unique filenames tmpnam() can generate.

There are three variables for the standard I/O streams:

```
extern FILE *stderr, *stdin, *stdout;
```

## stdlib.h

Besides a redefinition of `NULL`, `size_t` and `wchar_t`, this header file contains the type definitions listed in Table 15.10.

**Table 15.10** Type definitions in *stdlib.h*

Type Definition	Description
typedef div_t;	A struct for the return value of <code>div()</code> .
typedef ldiv_t;	A struct for the return value of <code>ldiv()</code> .

Table 15.11 lists the constants defined in *stdlib.h*

**Table 15.11** Constants defined in *stdlib.h*

Constant	Definition
EXIT_FAILURE	Exit code for unsuccessful termination.
EXIT_SUCCESS	Exit code for successful termination.
RAND_MAX	Maximum return value of <code>rand()</code> .
MB_LEN_MAX	Maximum number of bytes in a multi-byte character.

## time.h

This header file defines types and constants for time management. See Listing 15.2.

### Listing 15.2 time.h—Type definitions and constants

---

```
typedef clock_t;
typedef time_t;

struct tm {
    int tm_sec;      /* Seconds */
    int tm_min;     /* Minutes */
    int tm_hour;    /* Hours */
    int tm_mday;    /* Day of month: 0 .. 31 */
    int tm_mon;     /* Month: 0 .. 11 */
    int tm_year;    /* Year since 1900 */
    int tm_wday;    /* Day of week: 0 .. 6 (Sunday == 0) */
    int tm_yday;    /* day of year: 0 .. 365 */
    int tm_isdst;   /* Daylight saving time flag:
                    > 0 It is DST
                    0 It is not DST
                    < 0 unknown */
};
```

---

The constant `CLOCKS_PER_SEC` gives the number of clock ticks per second.

## string.h

The `string.h` file defines only functions to manipulate ANSI-C string but not types or special defines.

The functions are explained below together with all other ANSI functions.

## assert.h

The `assert.h` file defines the `assert()` macro. If the `NDEBUG` macro is defined, then `assert` does nothing. Otherwise, `assert()` calls the auxiliary function `_assert` if the sole macro parameter of `assert()` evaluates to 0 (`FALSE`). See Listing 15.3.

### Listing 15.3 Use `assert()` to assist in debugging

---

```
#ifndef NDEBUG
#define assert(EX)
```

---

## Types and Macros in the Standard Library

*stdarg.h*

---

```
#else
#define assert(EX) ((EX) ? 0 : _assert(__LINE__, __FILE__))
#endif
```

---

## stdarg.h

The `stdarg.h` file defines the type `va_list` and the macros `va_arg()`, `va_end()`, and `va_start()`. The type `va_list` implements a pointer to one argument of an open parameter list. The macro `va_start()` initializes a variable of type `va_list` to point to the first open parameter, given the last explicit parameter and its type as arguments. The macro `va_arg()` returns one open parameter, given its type and also makes the `va_list` argument pointing to the next parameter. The `va_end()` macro finally releases the actual pointer. For all implementations, the `va_end()` macro does nothing because `va_list` is implemented as an elementary data type and it must therefore not be released. The `va_start()` and the `va_arg()` macros have a type parameter, which is accessed only with `sizeof()`.

### Listing 15.4 Example using `stdarg.h`

---

```
char sum(long p, ...) {
    char res=0;
    va_list list= va_start()(p, long);
    res= va_arg(list, int); // (*)
    va_end(list);
    return res;
}

void main(void) {
    char c = 2;
    if (f(10L, c) != 2) Error();
}
```

---

In the line (\*) `va_arg` must be called with `int`, not with `char`. Because of the default argument-promotion rules of C, for integral types at least an `int` is passed and for floating types at least a `double` is passed. In other words, the result of using `va_arg(..., char)` or `va_arg(..., short)` is undefined in C. Especially be careful when using variables instead of types for `va_arg()`. In the example above “`res= va_arg(list, res)`” would not be correct unless `res` would have the type `int` and not `char`.



---

## ctype.h

The `ctype.h` file defines functions to check properties of characters, as if a character is a digit - `isdigit()`, a space - `isspace()`, and many others. These functions are either implemented as macros, or as real functions. The macro version is used when the `-Ot` compiler option is used or the macro `__OPTIMIZE_FOR_TIME__` is defined. The macros use a table called `_ctype`, whose length is 257 bytes. In this array, all properties tested by the various functions are encoded by single bits, taking the character as indices into the array. The function implementations otherwise do not use this table. They save memory by using the shorter call to the function (compared with the expanded macro).

The functions in Listing 15.5 are explained below together with all other ANSI functions.

### Listing 15.5 Macros defined in `ctypes.h`

---

```
extern unsigned char  _ctype[];
#define  _U  (1<<0)    /* Upper case      */
#define  _L  (1<<1)    /* Lower case      */
#define  _N  (1<<2)    /* Numeral (digit) */
#define  _S  (1<<3)    /* Spacing character */
#define  _P  (1<<4)    /* Punctuation     */
#define  _C  (1<<5)    /* Control character */
#define  _B  (1<<6)    /* Blank           */
#define  _X  (1<<7)    /* hexadecimal digit */

#ifdef __OPTIMIZE_FOR_TIME__ /* -Ot defines this macro */
#define  isalnum(c)  (_ctype[(unsigned char)(c+1)] & (_U|_L|_N))
#define  isalpha(c)  (_ctype[(unsigned char)(c+1)] & (_U|_L))
#define  iscntrl(c)  (_ctype[(unsigned char)(c+1)] & _C)
#define  isdigit(c)  (_ctype[(unsigned char)(c+1)] & _N)
#define  isgraph(c)  (_ctype[(unsigned char)(c+1)] & (_P|_U|_L|_N))
#define  islower(c)  (_ctype[(unsigned char)(c+1)] & _L)
#define  isprint(c)  (_ctype[(unsigned char)(c+1)] & (_P|_U|_L|_N|_B))
#define  ispunct(c)  (_ctype[(unsigned char)(c+1)] & _P)
#define  isspace(c)  (_ctype[(unsigned char)(c+1)] & _S)
#define  isupper(c)  (_ctype[(unsigned char)(c+1)] & _U)
#define  isxdigit(c) (_ctype[(unsigned char)(c+1)] & _X)
#define  tolower(c)  (isupper(c) ? ((c) - 'A' + 'a') : (c))
#define  toupper(c)  (islower(c) ? ((c) - 'a' + 'A') : (c))
#define  isascii(c)  (!(c) & ~127)
#define  toascii(c)  (c & 127)
#endif /* __OPTIMIZE_FOR_TIME__ */
```

---

## **Types and Macros in the Standard Library**

*ctype.h*

---

# The Standard Functions

This section describes all the standard functions in the ANSI-C library. Each function description contains the subsections listed in Table 16.1.

**Table 16.1** Function description subsections

Subsection	Description
Syntax	Shows the function's prototype and also which header file to include.
Description	A description of how to use the function.
Return	Describes what the function returns in which case. If the <code>errno</code> global variable is modified by the function, possible values are also described.
See also	Contains cross-references to related functions.

Functions not implemented because the implementation would be hardware-specific anyway (e.g., `clock()`) are marked by:

*Hardware specific*



appearing in the right margin next to the function's name. Functions for file I/O, which also depend on the particular hardware's setup and therefore also are not implemented, are marked by:

*File I/O*



in the right margin.

### **abort()**

#### **Syntax**

```
#include <stdlib.h>

void abort(void);
```

#### **Description**

`abort()` terminates the program. It does the following (in this order):

- raises signal `SIGABRT`
- flushes all open output streams
- closes all open files
- removes all temporary files
- calls `HALT`

If your application handles `SIGABRT` and the signal handler does not return (e.g., because it does a `longjmp()`), the application is not halted.

#### **See also**

`atexit()`  
`exit()`  
`raise()`  
`signal()`

### **abs()**

#### **Syntax**

```
#include <stdlib.h>

int abs(int i);
```

#### **Description**

`abs()` computes the absolute value of `i`.

#### **Return**

The absolute value of `i`; i.e., `i` if `i` is positive and `-i` if `i` is negative. If `i` is `-32,768`, this value is returned and `errno` is set to `ERANGE`.

#### **See also**

`fabs()` and `fabsf()`  
`labs()`

### **acos() and acosf()**

#### **Syntax**

```
#include <math.h>

double acos (double x);
float  acosf (float x);
```

#### **Description**

`acos()` computes the principal value of the arc cosine of  $x$ .

#### **Return**

The arc cosine  $\cos^{-1}(x)$  of  $x$  in the range between 0 and  $\pi$  if  $x$  is in the range  $-1 \leq x \leq 1$ . If  $x$  is not in this range, `NAN` is returned and `errno` is set to `EDOM`.

#### **See also**

`asin()` and `asinf()`  
`atan()` and `atanf()`  
`atan2()` and `atan2f()`  
`cos()` and `cosf()`  
`sin()` and `sinf()`  
`tan()` and `tanf()`

### asctime()

*Hardware  
specific*



#### Syntax

```
#include <time.h>

char * asctime(const struct tm* timeptr);
```

#### Description

asctime() converts the time, broken down in timeptr, into a string.

#### Return

A pointer to a string containing the time string.

#### See also

localtime()  
mktime()  
time()

### asin() and asinf()

#### Syntax

```
#include <math.h>

double asin(double x);
float  asinf(float x);
```

#### Description

`asin()` computes the principal value of the arc sine of  $x$ .

#### Return

The arc sine  $\sin^{-1}(x)$  of  $x$  in the range between  $-\pi/2$  and  $\pi/2$  if  $x$  is in the range  $-1 \leq x \leq 1$ . If  $x$  is not in this range, NAN is returned and `errno` is set to `EDOM`.

#### See also

`acos()` and `acosf()`  
`atan()` and `atanf()`  
`atan2()` and `atan2f()`  
`cos()` and `cosf()`  
`tan()` and `tanf()`



### **assert()**

#### **Syntax**

```
#include <assert.h>

void assert(int expr);
```

#### **Description**

`assert()` is a macro that indicates expression `expr` is expected to be true at this point in the program. If `expr` is false (0), `assert()` halts the program.

Compiling with option `-DNDEBUG` or placing the preprocessor control statement `#define NDEBUG`

before the `#include <assert.h>` statement effectively deletes all assertions from the program.

#### **See also**

`abort()`  
`exit()`

### atan() and atanf()

#### Syntax

```
#include <math.h>

double atan (double x);
float  atanf(float x);
```

#### Description

`atan()` computes the principal value of the arc tangent of  $x$ .

#### Return

The arc tangent  $\tan^{-1}(x)$ , in the range from  $-\pi/2$  to  $\pi/2$  rad.

#### See also

`acos()` and `acosf()`,  
`asin()` and `asinf()`,  
`atan2()` and `atan2f()`,  
`cos()` and `cosf()`,  
`sin()` and `sinf()`, and  
`tan()` and `tanf()`

## atan2() and atan2f()

### Syntax

```
#include <math.h>

double atan2(double y, double x);
float  atan2f(float y, float x);
```

### Description

`atan2()` computes the principal value of the arc tangent of  $y/x$ . It uses the sign of both operands to determine the quadrant of the result.

### Return

The arc tangent  $\tan^{-1}(y/x)$ , in the range from  $-\pi$  to  $\pi$  rad, if not both  $x$  and  $y$  are 0. If both  $x$  and  $y$  are 0, it returns 0.

### See also

```
acos() and acosf()
asin() and asinf()
atan() and atanf()
cos() and cosf()
sin() and sinf()
tan() and tanf()
```

### **atexit()**

#### **Syntax**

```
#include <stdlib.h>

int atexit(void (*func) (void));
```

#### **Description**

`atexit()` lets you install a function that is to be executed just before the normal termination of the program. You can register at most 32 functions with `atexit()`. These functions are called in the reverse order they were registered.

#### **Return**

`atexit()` returns 0 if it could register the function, otherwise it returns a non-zero value.

#### **See also**

`abort()`  
`exit()`

## atof()

### Syntax

```
#include <stdlib.h>

double atof(const char *s);
```

### Description

`atof()` converts the string `s` to a double floating point value, skipping over white space at the beginning of `s`. It stops converting when it reaches either the end of the string or a character that cannot be part of the number. The number format accepted by `atof()` is as follows:

```
FloatNum    = Sign{Digit}[.{Digit}][Exp]
Sign        = [+|-]
Digit       = <any decimal digit from 0 to 9>
Exp         = (e|E) SignDigit{Digit}
```

### Return

`atof()` returns the converted double floating point value.

### See also

- `atoi()`
- `strtod()`
- `strtol()`
- `strtoul()`

### atoi()

#### Syntax

```
#include <stdlib.h>

int atoi(const char *s);
```

#### Description

`atoi()` converts the string `s` to an integer value, skipping over white space at the beginning of `s`. It stops converting when it reaches either the end of the string or a character that cannot be part of the number. The number format accepted by `atoi()` is as follows:

Number = [+|-]Digit{Digit}

#### Return

`atoi()` returns the converted integer value.

#### See also

- `atof()`
- `atol()`
- `strtod()`
- `strtol()`
- `strtoul()`

## atol()

### Syntax

```
#include <stdlib.h>

long atol(const char *s);
```

### Description

`atol()` converts the string `s` to an `long` value, skipping over white space at the beginning of `s`. It stops converting when it reaches either the end of the string or a character that cannot be part of the number. The number format accepted by `atol()` is as follows:

```
Number = [+|-]Digit{Digit}
```

### Return

`atol()` returns the converted `long` value.

### See also

- `atoi()`
- `atof()`
- `strtod()`
- `strtol()`
- `strtoul()`

### bsearch()

#### Syntax

```
#include <stdlib.h>

void *bsearch(const void *key,
             const void *array,
             size_t n,
             size_t size,
             cmp_func cmp());
```

#### Description

`bsearch()` performs a binary search in a sorted array. It calls the comparison function `cmp()` with two arguments: a pointer to the key element that is to be found and a pointer to an array element. Thus, the `cmp_func` type can be declared as:

```
typedef int (*cmp_func)(const void *key,
                       const void *data);
```

The comparison function should return an integer according to (Table 16.2):

**Table 16.2** Return value from the comparison function, `cmp_func()`

If the key element is...	the return value should be...
less than the array element	less than zero (negative)
equal to the array element	zero
greater than the array element	greater than zero (positive)



The arguments (Table 16.3) of `bsearch()` are:

**Table 16.3 Possible arguments to the `bsearch()` function**

Parameter Name	Meaning
<code>key</code>	A pointer to the key data you are seeking
<code>array</code>	A pointer to the beginning (i.e., the first element) of the array that is searched
<code>n</code>	The number of elements in the array
<code>size</code>	The size (in bytes) of one element in the table
<code>cmp()</code>	The comparison function

---

**NOTE** Make sure the array contains only elements of the same size. `bsearch()` also assumes that the array is sorted in ascending order with respect to the comparison function `cmp()`.

---

### Return

`bsearch()` returns a pointer to an element of the array that matches the key, if there is one. If the comparison function never returns zero, i.e., there is no matching array element, `bsearch()` returns `NULL`.

### **calloc()**

*Hardware  
specific*



#### **Syntax**

```
#include <stdlib.h>

void *calloc(size_t n, size_t size);
```

#### **Description**

`calloc()` allocates a block of memory for an array containing `n` elements of size `size`. All bytes in the memory block are initialized to zero. To deallocate the block, use `free()`. The default implementation is not reentrant and should therefore not be used in interrupt routines.

#### **Return**

`calloc()` returns a pointer to the allocated memory block. If the block could not be allocated, the return value is `NULL`.

#### **See also**

`malloc()`  
`realloc()`

### **ceil() and ceilf()**

#### **Syntax**

```
#include <math.h>

double ceil(double x);
float  ceilf(float x);
```

#### **Description**

`ceil()` returns the smallest integral number larger than `x`.

#### **See also**

`floor()` and `floorf()`  
`fmod()` and `fmodf()`

### clearerr()

*File I/O*



#### Syntax

```
#include <stdio.h>

void clearerr(FILE *f);
```

#### Description

`clearerr()` resets the error flag and the EOF marker of file `f`.

### clock()

*Hardware  
specific*



#### Syntax

```
#include <time.h>

clock_t clock(void);
```

#### Description

`clock()` determines the amount of time since your system started, in clock ticks. To convert to seconds, divide by `CLOCKS_PER_SEC`.

#### Return

`clock()` returns the amount of time since system startup.

#### See also

`time()`

### cos() and cosf()

#### Syntax

```
#include <time.h>

double cos (double x);
float  cosf(float x);
```

#### Description

`cos()` computes the principal value of the cosine of `x`. `x` should be expressed in radians.

#### Return

The cosine `cos(x)`.

#### See also

`acos()` and `acosf()`  
`asin()` and `asinf()`  
`atan()` and `atanf()`  
`atan2()` and `atan2f()`  
`sin()` and `sinf()`  
`tan()` and `tanf()`

### **cosh() and coshf()**

#### **Syntax**

```
#include <time.h>

double cosh (double x);
float  coshf(float x);
```

#### **Description**

`cosh()` computes the hyperbolic cosine of `x`.

#### **Return**

The hyperbolic cosine `cosh(x)`. If the computation fails because the value is too large, `HUGE_VAL` is returned and `errno` is set to `ERANGE`.

#### **See also**

`cos()` and `cosf()`  
`sinh()` and `sinhf()`  
`tanh()` and `tanhf()`

### ctime()

*Hardware  
specific*



#### Syntax

```
#include <time.h>

char *ctime(const time_t *timer);
```

#### Description

`ctime()` converts the calendar time `timer` to a character string.

#### Return

The string containing the ASCII representation of the date.

#### See also

`asctime()`  
`mktime()`  
`time()`



### **difftime()**

*Hardware  
specific*



#### **Syntax**

```
#include <time.h>

double difftime(time_t *t1, time_t t0);
```

#### **Description**

`difftime()` calculates the number of seconds between any two calendar times.

#### **Return**

The number of seconds between the two times, as a `double`.

#### **See also**

`mktime()`  
`time()`

### div()

#### Syntax

```
#include <stdlib.h>

div_t div(int x, int y);
```

#### Description

`div()` computes both the quotient and the modulus of the division  $x/y$ .

#### Return

A structure with the results of the division.

#### See also

`ldiv()`

### **exit()**

#### **Syntax**

```
#include <stdlib.h>

void exit(int status);
```

#### **Description**

`exit()` terminates the program normally. It does the following, in this order:

- executes all functions registered with `atexit()`
- flushes all open output streams
- closes all open files
- removes all temporary files
- calls `HALT`

The `status` argument is ignored.

#### **See also**

`abort()`

### exp() and expf()

#### Syntax

```
#include <math.h>

double exp (double x);
float  expf(float x);
```

#### Description

exp () computes  $e^x$ , where  $e$  is the base of natural logarithms.

#### Return

$e^x$ . If the computation fails because the value is too large, HUGE\_VAL is returned and `errno` is set to ERANGE.

#### See also

log() and logf()  
log10() and log10f()  
pow() and powf()

### **fabs() and fabsf()**

#### **Syntax**

```
#include <math.h>

double fabs (double x);
float  fabsf(float x);
```

#### **Description**

`fabs()` computes the absolute value of `x`.

#### **Return**

The absolute value of `x` for any value of `x`.

#### **See also**

`abs()`  
`labs()`

### **fclose()**

*File I/O*



#### **Syntax**

```
#include <stdlib.h>

int fclose(FILE *f);
```

#### **Description**

`fclose()` closes file `f`. Before doing so, it does the following:

- flushes the stream, if the file was not opened in read-only mode
- discards and deallocates any buffers that were allocated automatically, i.e., not using `setbuf()`.

#### **Return**

Zero, if the function succeeds; EOF otherwise.

#### **See also**

`fopen()`

### **feof()**

*File I/O*



#### **Syntax**

```
#include <stdio.h>

int feof(FILE *f);
```

#### **Description**

`feof()` tests whether previous I/O calls on file `f` tried to do anything beyond the end of the file.

---

**NOTE** Calling `clearerr()` or `fseek()` clears the file's end-of-file flag; therefore `feof()` returns 0.

---

#### **Return**

Zero, if not at the end of the file; EOF otherwise.

### **ferror()**

*File I/O*



#### **Syntax**

```
#include <stdio.h>

int ferror(FILE *f);
```

#### **Description**

`ferror()` tests whether an error had occurred on file `f`. To clear the error indicator of a file, use `clearerr()`. `rewind()` automatically resets the file's error flag.

---

**NOTE** Do not use `ferror()` to test for end-of-file. Use `feof()` instead.

---

#### **Return**

Zero, if there was no error; non-zero otherwise.



**fflush()***File I/O***Syntax**

```
#include <stdio.h>

int fflush(FILE *f);
```

**Description**

`fflush()` flushes the I/O buffer of file `f`, allowing a clean switch between reading and writing the same file. If the program was writing to file `f`, `fflush()` writes all buffered data to the file. If it was reading, `fflush()` discards any buffered data. If `f` is `NULL`, *all* files open for writing are flushed.

**Return**

Zero, if there was no error; EOF otherwise.

**See also**

`setbuf()`  
`setvbuf()`

### fgetc()

File I/O



#### Syntax

```
#include <stdio.h>

int fgetc(FILE *f);
```

#### Description

`fgetc()` reads the next character from file `f`.

---

**NOTE** If file `f` had been opened as a text file, the end-of-line character combination is read as one `'\n'` character.

---

#### Return

The character is read as an integer in the range from 0 to 255. If there was a read error, `fgetc()` returns `EOF` and sets the file's error flag, so that a subsequent call to `ferror()` will return a non-zero value. If an attempt is made to read beyond the end of the file, `fgetc()` also returns `EOF`, but sets the end-of-file flag instead of the error flag so that `feof()` will return `EOF`, but `ferror()` will return 0.

#### See also

- `fgets()`
- `fopen()`
- `fread()`
- `fscanf()`
- `getc()`

**fgetpos()**

File I/O

**Syntax**

```
#include <stdio.h>

int fgetpos(FILE *f, fpos_t *pos);
```

**Description**

`fgetpos()` returns the current file position in `*pos`. This value can be used to later set the position to this one using `fsetpos()`.

---

**NOTE** Do *not* assume the value in `*pos` to have any particular meaning such as a byte offset from the beginning of the file. The ANSI standard does not require this, and in fact any value may be put into `*pos` as long as there is a `fsetpos()` with that value resets the position in the file correctly.

---

**Return**

Non-zero, if there was an error; zero otherwise.

**See also**

`fseek()`  
`ftell()`

### fgets()

*File I/O*



#### Syntax

```
#include <stdio.h>

char *fgets(char *s, int n, FILE *f);
```

#### Description

`fgets()` reads a string of at most  $n-1$  characters from file `f` into `s`. Immediately after the last character read, a `'\0'` is appended. If `fgets()` reads a line break (`'\n'`) or reaches the end of the file before having read  $n-1$  characters, the following happens:

- If `fgets()` reads a line break, it adds the `'\n'` plus a `'\0'` to `s` and returns successfully.
- If it reaches the end of the file after having read at least 1 character, it adds a `'\0'` to `s` and returns successfully.
- If it reaches EOF without having read any character, it sets the file's end-of-file flag and returns unsuccessfully. (`s` is left unchanged.)

#### Return

NULL, if there was an error; `s` otherwise.

#### See also

`fgetc()`  
`fputs()`

### **floor() and floorf()**

#### **Syntax**

```
#include <math.h>

double floor (double x);
float  floorf(float x);
```

#### **Description**

`floor()` calculates the largest integral number not larger than `x`.

#### **Return**

The largest integral number not larger than `x`.

#### **See also**

`ceil()` and `ceilf()`  
`modf()` and `modff()`

### fmod() and fmodf()

#### Syntax

```
#include <math.h>

double fmod (double x, double y);
float fmodf(float x, float y);
```

#### Description

`fmod()` calculates the floating point remainder of  $x/y$ .

#### Return

The floating point remainder of  $x/y$ , with the same sign as  $x$ . If  $y$  is 0, it returns 0 and sets `errno` to `EDOM`.

#### See also

`div()`  
`ldiv()`  
`ldexp()` and `ldexpf()`  
`modf()` and `modff()`

**fopen()***File I/O***Syntax**

```
#include <stdio.h>

FILE *fopen(const char *name, const char *mode);
```

**Description**

`fopen()` opens a file with the given name and mode. It automatically allocates an I/O buffer for the file.

There are three main modes: read, write, and update (i.e., both read and write) accesses. Each can be combined with either text or binary mode to read a text file or update a binary file. Opening a file for text accesses translates the end-of-line character (combination) into '`\n`' when reading and vice versa when writing. Table 16.4 lists all possible modes.

**Table 16.4 Operating modes of the file opening function, `fopen()`**

Mode	Effect
r	Open the file as a text file for reading.
w	Create a text file and open it for writing.
a	Open the file as a text file for appending
rb	Open the file as a binary file for reading.
wb	Create a file and open as a binary file for writing.
ab	Open the file as a binary file for appending.
r+	Open a text file for updating.
w+	Create a text file and open for updating.
a+	Open a text file for updating. Append all writes to the end.
r+b or rb+	Open a binary file for updating.
w+b or wb+	Create a binary file and open for updating.
a+b or ab+	Open a binary file for updating, appending all writes to the end.

## The Standard Functions

---

If the mode contains an “r”, but the file doesn’t exist, `fopen()` returns unsuccessfully. Opening a file for appending (mode contains “a”) always appends writing to the end, even if `fseek()`, `fsetpos()`, or `rewind()` is called. Opening a file for updating allows both read and write accesses on the file. However, `fseek()`, `fsetpos()`, or `rewind()` must be called in order to write after a read or to read after a write.

### Return

A pointer to the file descriptor of the file. If the file could not be created, the function returns `NULL`.

### See also

- `fclose()`
- `freopen()`
- `setbuf()`
- `setvbuf()`



### **fprintf()**

#### **Syntax**

```
#include <stdio.h>

int fprintf(FILE *f, const char *format, ...);
```

#### **Description**

`fprintf()` is the same as `sprintf()`, but the output goes to file `f` instead of a string.

For a detailed format description, see `sprintf()`.

#### **Return**

The number of characters written. If some error occurred, EOF is returned.

#### **See also**

`printf()`  
`vsprintf()`

### fputc()

*File I/O*



#### Syntax

```
#include <stdio.h>

int fputc(int ch, FILE *f);
```

#### Description

`fputc()` writes a character to file `f`.

#### Return

The integer value of `ch`. If an error occurred, `fputc()` returns EOF.

#### See also

`fputs()`

### **fputs()**

*File I/O*



#### **Syntax**

```
#include <stdio.h>

int fputs(const char *s, FILE *f);
```

#### **Description**

`fputs()` writes the zero-terminated string `s` to file `f` (without the terminating `'\0'`).

#### **Return**

EOF, if there was an error; zero otherwise.

#### **See also**

`fputc()`

### fread()

*File I/O*



#### Syntax

```
#include <stdio.h>
```

```
size_t fread(void *ptr, size_t size, size_t n, FILE *f);
```

#### Description

`fread()` reads a contiguous block of data. It attempts to read `n` items of size `size` from file `f` and stores them in the array to which `ptr` points. If either `n` or `size` is 0, nothing is read from the file and the array is left unchanged.

#### Return

The number of items successfully read.

#### See also

`fgetc()`

`fgets()`

`fwrite()`

### **free()**

*Hardware  
specific*



#### **Syntax**

```
#include <stdlib.h>

void free(void *ptr);
```

#### **Description**

`free()` deallocates a memory block that had previously been allocated by `calloc()`, `malloc()`, or `realloc()`. If `ptr` is `NULL`, nothing happens. The default implementation is not reentrant and should therefore not be used in interrupt routines.

### freopen()

*File I/O*



#### Syntax

```
#include <stdio.h>

void freopen(const char *name,
            const char *mode,
            FILE *f);
```

#### Description

`freopen()` opens a file using a specific file descriptor. This can be useful for redirecting `stdin`, `stdout`, or `stderr`. About possible modes, see `fopen()`.

#### See also

`fclose()`

## frexp() and frexpf()

### Syntax

```
#include <math.h>

double frexp(double x, int *exp);
float  frexpf(float x, int *exp);
```

### Description

`frexp()` splits a floating point number into mantissa and exponent. The relation is  $x = m * 2^{\text{exp}}$ .  $m$  always is normalized to the range  $0.5 < m \leq 1.0$ . The mantissa has the same sign as  $x$ .

### Return

The mantissa of  $x$  (the exponent is written to `*exp`). If  $x$  is `0.0`, both the mantissa (the return value) and the exponent are `0`.

### See also

`exp()` and `expf()`  
`ldexp()` and `ldexpf()`  
`modf()` and `modff()`

### **fscanf()**

*File I/O*



#### **Syntax**

```
#include <stdio.h>

int fscanf(FILE *f, const char *format, ...);
```

#### **Description**

`fscanf()` is the same as `scanf()` but the input comes from file `f` instead of a string.

#### **Return**

The number of data arguments read, if any input was converted. If not, it returns EOF.

#### **See also**

`fgetc()`  
`fgets()`  
`sscanf()`



**fseek()**

File I/O

**Syntax**

```
#include <stdio.h>

int fseek(FILE *f, long offset, int mode);
```

**Description**

`fseek()` sets the current position in file `f`.

For binary files, the position can be set in three ways, as shown in this table.

**Table 16.5 Offset position into the file for the `fseek()` function**

<b>mode</b>	<b>Position is set to...</b>
SEEK_SET	offset bytes from the beginning of the file.
SEEK_CUR	offset bytes from the current position.
SEEK_END	offset bytes from the end of the file.

For text files, either `offset` must be zero or `mode` is `SEEK_SET` and `offset` a value returned by a previous call to `ftell()`.

If `fseek()` is successful, it clears the file's end-of-file flag. The position cannot be set beyond the end of the file.

**Return**

Zero, if successful; non-zero otherwise.

**See also**

`fgetpos()`  
`fsetpos()`  
`ftell()`

### **fsetpos()**

*File I/O*



#### **Syntax**

```
#include <stdio.h>

int fsetpos(FILE *f, const fpos_t *pos);
```

#### **Description**

`fsetpos()` sets the file position to `pos`, which must be a value returned by a previous call to `fgetpos()` on the same file. If the function is successful, it clears the file's end-of-file flag.

The position cannot be set beyond the end of the file.

#### **Return**

Zero, if it was successful; non-zero otherwise.

#### **See also**

`fgetpos()`  
`fseek()`  
`ftell()`

### **ftell()**

*File I/O*



#### **Syntax**

```
#include <stdio.h>

long ftell(FILE *f);
```

#### **Description**

`ftell()` returns the current file position. For binary files, this is the byte offset from the beginning of the file; for text files, this value should not be used except as argument to `fseek()`.

#### **Return**

-1, if an error occurred; otherwise the current file position.

#### **See also**

`fgetpos()`  
`fsetpos()`

### **fwrite()**

*File I/O*



#### **Syntax**

```
#include <stdio.h>

size_t fwrite(const void *p,
              size_t size,
              size_t n,
              FILE *f);
```

#### **Description**

`fwrite()` writes a block of data to file `f`. It writes `n` items of size `size`, starting at address `ptr`.

#### **Return**

The number of items successfully written.

#### **See also**

`fputc()`  
`fputs()`  
`fread()`

### **getc()**

*File I/O*



#### **Syntax**

```
#include <stdio.h>

int getc(FILE *f);
```

#### **Description**

`getc()` is the same as `fgetc()`, but may be implemented as a macro. Therefore, make sure that `f` is not an expression having side effects! See `fgetc()` for more information.

### **getchar()**

*File I/O*



#### **Syntax**

```
#include <stdio.h>
int getchar(void);
```

#### **Description**

`getchar()` is the same as `getc(stdin)`. See `fgetc()` for more information.

## getenv()

*File I/O*



### Syntax

```
#include <stdio.h>

char *getenv(const char *name);
```

### Description

`getenv()` returns the value of environment variable `name`.

### Return

NULL

### gets()

*File I/O*



#### Syntax

```
#include <stdio.h>

char *gets(char *s);
```

#### Description

`gets()` reads a string from `stdin` and stores it in `s`. It stops reading when it reaches a line break or EOF character. This character is not appended to the string. The string is zero-terminated.

If the function reads EOF before any other character, it sets `stdin`'s end-of-file flag and returns unsuccessfully without changing string `s`.

#### Return

NULL, if there was an error; `s` otherwise.

#### See also

`fgetc()`  
`puts()`



### gmtime()

*Hardware  
specific*



#### Syntax

```
#include <time.h>

struct tm *gmtime(const time_t *time);
```

#### Description

`gmtime()` converts `*time` to UTC (Universal Coordinated Time), which is equivalent to GMT (Greenwich Mean Time).

#### Return

`NULL`, if UTC is not available; a pointer to a struct containing UTC otherwise.

#### See also

`ctime()`  
`time()`

### **isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), and isxdigit()**

#### **Syntax**

```
#include <ctype.h>

int isalnum (int ch);
int isalpha (int ch);
...
int isxdigit(int ch);
```

#### **Description**

These functions determine whether character `ch` belongs to a certain set of characters. The next table describes the character ranges tested by the functions.

**Table 16.6 Appropriate character range for the testing functions**

<b>Function</b>	<b>Tests whether <code>ch</code> is in the range...</b>
<code>isalnum()</code>	alphanumeric character, i.e., 'A'-'Z', 'a'-'z' or '0'-'9'.
<code>isalpha()</code>	an alphabetic character, i.e., 'A'-'Z' or 'a'-'z'.
<code>iscntrl()</code>	a control character, i.e., '\000'-'037' or '\177' (DEL).
<code>isdigit()</code>	a decimal digit, i.e., '0'-'9'.
<code>isgraph()</code>	a printable character except space ('!'-'~').
<code>islower()</code>	a lower case letter, i.e., 'a'-'z'.
<code>isprint()</code>	a printable character (' '-'~').
<code>ispunct()</code>	a punctuation character, i.e., '!'-'/', ':', '@', '[', '-' and '{'-'~'.
<code>isspace()</code>	a white space character, i.e., ' ', '\f', '\n', '\r', '\t' and '\v'.
<code>isupper()</code>	an upper case letter, i.e., 'A'-'Z'.
<code>isxdigit()</code>	a hexadecimal digit, i.e., '0'-'9', 'A'-'F' or 'a'-'f'.

### Return

TRUE (i.e., 1), if `ch` is in the character class; zero otherwise.

### See also

`tolower()`

`toupper()`

### labs()

#### Syntax

```
#include <stdlib.h>

long labs(long i);
```

#### Description

`labs()` computes the absolute value of `i`.

#### Return

The absolute value of `i`, i.e., `i` if `i` is positive and `-i` if `i` is negative. If `i` is `-2,147,483,648`, this value is returned and `errno` is set to `ERANGE`.

#### See also

`abs()`

## ldexp() and ldexpf()

### Syntax

```
#include <math.h>

double ldexp (double x, int exp);
float ldexpf(float x, int exp);
```

### Description

ldexp() multiplies  $x$  by  $2^{\text{exp}}$ .

### Return

$x * 2^{\text{exp}}$ . If it fails because the result would be too large, HUGE\_VAL is returned and `errno` is set to ERANGE.

### See also

exp() and expf()  
frexp() and frexpf()  
log() and logf()  
log10() and log10f()  
modf() and modff()

### ldiv()

#### Syntax

```
#include <stdlib.h>

ldiv_t ldiv(long x, long y);
```

#### Description

`ldiv()` computes both the quotient and the modulus of the division  $x/y$ .

#### Return

A structure with the results of the division.

#### See also

`div()`

### localeconv()

*Hardware  
specific*



#### Syntax

```
#include <locale.h>

struct lconv *localeconv(void);
```

#### Description

`localeconv()` returns a pointer to a `struct` containing information about the current locale, e.g., how to format monetary quantities.

#### Return

A pointer to a `struct` containing the desired information.

#### See also

`setlocale()`

### localtime()

*Hardware  
specific*



#### Syntax

```
#include <time.h>

struct tm *localtime(const time_t *time);
```

#### Description

`localtime()` converts `*time` into broken-down time.

#### Return

A pointer to a `struct` containing the broken-down time.

#### See also

`asctime()`  
`mktime()`  
`time()`



## log() and logf()

### Syntax

```
#include <math.h>

double log (double x);
float  logf(float x);
```

### Description

`log()` computes the natural logarithm of  $x$ .

### Return

$\ln(x)$ , if  $x$  is greater than zero. If  $x$  is smaller than zero, NAN is returned; if it is equal to zero, `log()` returns negative infinity. In both cases, `errno` is set to EDOM.

### See also

`exp()` and `expf()`  
`log10()` and `log10f()`

### log10() and log10f()

#### Syntax

```
#include <math.h>

double log10(double x);
float  log10f(float x);
```

#### Description

`log10()` computes the decadic logarithm (the logarithm to base 10) of `x`.

#### Return

`log10(x)`, if `x` is greater than zero. If `x` is smaller than zero, `NAN` is returned; if it is equal to zero, `log10()` returns negative infinity. In both cases, `errno` is set to `EDOM`.

#### See also

`exp()` and `expf()`  
`log10()` and `log10f()`

### **longjmp()**

#### **Syntax**

```
#include <setjmp.h>

void longjmp(jmp_buf env, int val);
```

#### **Description**

`longjmp()` performs a non-local jump to some location earlier in the call chain. That location must have been marked by a call to `setjmp()`. The environment at the time of that call to `setjmp()` - `env`, which also was the parameter to `setjmp()` - is restored and your application continues as if the call to `setjmp()` just had returned the value `val`.

#### **See also**

`setjmp()`

### malloc()

*Hardware  
specific*



#### Syntax

```
#include <stdlib.h>

void *malloc(size_t size);
```

#### Description

`malloc()` allocates a block of memory for an object of size `size` bytes. The content of this memory block is undefined. To deallocate the block, use `free()`. The default implementation is not reentrant and should therefore not be used in interrupt routines.

#### Return

`malloc()` returns a pointer to the allocated memory block. If the block could not be allocated, the return value is `NULL`.

#### See also

`calloc()`  
`realloc()`

**mblen()***Hardware  
specific***Syntax**

```
#include <stdlib.h>

int mblen(const char *s, size_t n);
```

**Description**

`mblen()` determines the number of bytes the multi-byte character pointed to by `s` occupies.

**Return**

0, if `s` is NULL.  
-1, if the first `n` bytes of `*s` do not form a valid multi-byte character.  
`n`, the number of bytes of the multi-byte character otherwise.

**See also**

`mbtowc()`  
`mbstowcs()`

### mbstowcs()

*Hardware  
specific*



#### Syntax

```
#include <stdlib.h>

size_t mbstowcs(wchar_t *wcs,
                const char *mbs,
                size_t n);
```

#### Description

`mbstowcs()` converts a multi-byte character string `mbs` to a wide character string `wcs`. Only the first `n` elements are converted.

#### Return

The number of elements converted, or `(size_t) - 1` if there was an error.

#### See also

`mblen()`  
`mbtowc()`

### mbtowc()

*Hardware  
specific*



#### Syntax

```
#include <stdlib.h>

int mbtowc(wchar_t *wc, const char *s, size_t n);
```

#### Description

`mbtowc()` converts a multi-byte character `s` to a wide character code `wc`. Only the first `n` bytes of `*s` are taken into consideration.

#### Return

The number of bytes of the multi-byte character converted (`size_t`) if successful or `-1` if there was an error.

#### See also

`mblen()`  
`mbstowcs()`

### memchr()

#### Syntax

```
#include <string.h>

void *memchr(const void *p, int ch, size_t n);
```

#### Description

`memchr()` looks for the first occurrence of a byte containing (`ch & 0xFF`) in the first `n` bytes of the memory are pointed to by `p`.

#### Return

A pointer to the byte found, or `NULL` if no such byte was found.

#### See also

- `memcmp()`
- `strchr()`
- `strrchr()`



### memcmp()

#### Syntax

```
#include <string.h>

void *memcmp(const void *p,
             const void *q,
             size_t n);
```

#### Description

`memcmp()` compares the first `n` bytes of the two memory areas pointed to by `p` and `q`.

#### Return

A positive integer, if `p` is considered greater than `q`; a negative integer if `p` is considered smaller than `q` or zero if the two memory areas are equal.

#### See also

`memchr()`  
`strcmp()`  
`strncmp()`

### memcpy() and memmove()

#### Syntax

```
#include <string.h>

void *memcpy(const void *p,
             const void *q,
             size_t n);

void *memmove(const void *p,
              const void *q,
              size_t n);
```

#### Description

Both functions copy *n* bytes from *q* to *p*. `memmove()` also works if the two memory areas overlap.

#### Return

*p*

#### See also

`strcpy()`  
`strncpy()`

### memset()

#### Syntax

```
#include <string.h>

void *memset(void *p, int val, size_t n);
```

#### Description

memset () sets the first n bytes of the memory area pointed to by p to the value (val & 0xFF).

#### Return

p

#### See also

calloc()  
memmove()

### mktime()

*Hardware  
specific*



#### Syntax

```
#include <string.h>

time_t mktime(struct tm *time);
```

#### Description

`mktime()` converts `*time` to a `time_t`. The fields of `*time` may have any value; they are not restricted to the ranges given `time.h`. If the conversion was successful, `mktime()` restricts the fields of `*time` to these ranges and also sets the `tm_wday` and `tm_yday` fields correctly.

#### Return

`*time` as a `time_t`

#### See also

`ctime()`  
`gmtime()`  
`time()`

### modf() and modff()

#### Syntax

```
#include <math.h>

double modf(double x, double *i);
float  modff(float x, float *i);
```

#### Description

`modf()` splits the floating-point number `x` into an integral part (returned in `*i`) and a fractional part. Both parts have the same sign as `x`.

#### Return

The fractional part of `x`

#### See also

`floor()` and `floorf()`  
`fmod()` and `fmodf()`  
`frexp()` and `frexpf()`  
`ldexp()` and `ldexpf()`

### **perror()**

#### **Syntax**

```
#include <stdio.h>

void perror(const char *msg);
```

#### **Description**

`perror()` writes an error message appropriate for the current value of `errno` to `stderr`. The character string `msg` is part of `perror`'s output.

#### **See also**

`assert()`  
`strerror()`

## pow() and powf()

### Syntax

```
#include <math.h>

double pow (double x, double y);
float powf(float x, float y);
```

### Description

pow() computes  $x$  to the power of  $y$ , i.e.,  $x^y$ .

### Return

---

```
 $x^y$ ,    if  $x > 0$ 
1,      if  $y == 0$ 
+ $x$ ,    if ( $x == 0$  &&  $y < 0$ )
NaN,   if ( $x < 0$  &&  $y$  is not integral). Also, errno is set to EDOM.
 $\pm x$ ,  with the same sign as  $x$ , if the result is too large.
```

---

### See also

exp() and expf()  
ldexp() and ldexpf()  
log() and logf()  
modf() and modff()

### printf()

*File I/O*



#### Syntax

```
#include <stdio.h>

int printf(const char *format, ...);
```

#### Description

`printf()` is the same as `sprintf()`, but the output goes to `stdout` instead of a string.

For a detailed format description see `sprintf()`.

#### Return

The number of characters written. If some error occurred, EOF is returned.

#### See also

`fprintf()`  
`vsprintf()`



### putc()

*File I/O*



#### Syntax

```
#include <stdio.h>

int putc(char ch, FILE *f);
```

#### Description

`putc()` is the same as `fputc()`, but may be implemented as a macro. Therefore, you should make sure that `f` is not an expression having side effects! See `fputc()` for more information.

### putchar()

*File I/O*



#### Syntax

```
#include <stdio.h>

int putchar(char ch);
```

#### Description

`putchar(ch)` is the same as `putc(ch, stdin)`. See `fputc()` for more information.

### puts()

*File I/O*



#### Syntax

```
#include <stdio.h>

int puts(const char *s);
```

#### Description

`puts()` writes string `s` followed by a newline `'\n'` to `stdout`.

#### Return

EOF, if there was an error; zero otherwise.

#### See also

`fputc()`

`putc()`

### qsort()

#### Syntax

```
#include <stdlib.h>

void *qsort(const void *array, size_t n,
            size_t size, cmp_func cmp);
```

#### Description

`qsort()` sorts the array according to the ordering implemented by the comparison function. It calls the comparison function `cmp()` with two pointers to array elements. Thus, the type `cmp_func()` can be declared as:

```
typedef int (*cmp_func)(const void *key,
                       const void *other);
```

The comparison function should return an integer according to Table 16.7.

**Table 16.7 Return value from the comparison function, `cmp_func()`**

If the key element is...	The return value should be...
less than the other one	less than zero (negative)
equal to the other one	zero
greater than the other one	greater than zero (positive)

The arguments to `qsort()` are listed in Table 16.8.

**Table 16.8 Possible arguments to the sorting function, `qsort()`**

Argument Name	Meaning
<code>array</code>	A pointer to the beginning (i.e., the first element) of the array to be sorted
<code>n</code>	The number of elements in the array

---

**Table 16.8** Possible arguments to the sorting function, `qsort()` (*continued*)

Argument Name	Meaning
<code>size</code>	The size (in bytes) of one element in the table
<code>cmp()</code>	The comparison function

---

**NOTE** Make sure the array contains elements of the same size.

---

### **raise()**

#### **Syntax**

```
#include <signal.h>

int raise(int sig);
```

#### **Description**

`raise()` raises the given signal, invoking the signal handler or performing the defined response to the signal. If a response was not defined or a signal handler was not installed, the application is aborted.

#### **Return**

Non-zero, if there was an error; zero otherwise.

#### **See also**

`signal()`

### rand()

#### Syntax

```
#include <stdlib.h>

int rand(void);
```

#### Description

`rand()` generates a pseudo random number in the range from 0 to `RAND_MAX`. The numbers generated are based on a seed, which initially is 1. To change the seed, use `srand()`.

The same seeds always lead to the same sequence of pseudo random numbers.

#### Return

A pseudo random integer in the range from 0 to `RAND_MAX`.

### realloc()

*Hardware  
specific*



#### Syntax

```
#include <stdlib.h>

void *realloc(void *ptr, size_t size);
```

#### Description

`realloc()` changes the size of a block of memory, preserving its contents. `ptr` must be a pointer returned by `calloc()`, `malloc()`, `realloc()`, or `NULL`. In the latter case, `realloc()` is equivalent to `malloc()`.

If the new size of the memory block is smaller than the old size, `realloc()` discards that memory at the end of the block. If size is zero (and `ptr` is not `NULL`), `realloc()` frees the whole memory block.

If there is not enough memory to perform the `realloc()`, the old memory block is left unchanged, and `realloc()` returns `NULL`. The default implementation is not reentrant and should therefore not be used in interrupt routines.

#### Return

`realloc()` returns a pointer to the new memory block. If the operation could not be performed, the return value is `NULL`.

#### See also

`free()`



### **remove()**

*File I/O*



#### **Syntax**

```
#include <stdio.h>
int remove(const char *filename);
```

#### **Description**

`remove()` deletes the file `filename`. If the file is open, `remove()` does not delete it and returns unsuccessfully.

#### **Return**

Non-zero, if there was an error; zero otherwise.

#### **See also**

`tmpfile()`  
`tmpnam()`

### rename()

*File I/O*



#### Syntax

```
#include <stdio.h>

int rename(const char *from, const char *to);
```

#### Description

`rename()` renames the `from` file to `to`. If there already is a `to` file, `rename()` does not change anything and returns with an error code.

#### Return

Non-zero, if there was an error; zero otherwise.

#### See also

`tmpfile()`  
`tmpnam()`

### rewind()

*File I/O*



#### Syntax

```
#include <stdio.h>

void rewind(FILE *f);
```

#### Description

`rewind()` resets the current position in file `f` to the beginning of the file. It also clears the file's error indicator.

#### See also

`fopen()`  
`fseek()`  
`fsetpos()`

### scanf()

*File I/O*



#### Syntax

```
#include <stdio.h>

int scanf(const char *format, ...);
```

#### Description

`scanf()` is the same as `sscanf()`, but the input comes from `stdin` instead of a string.

#### Return

The number of data arguments read, if any input was converted. If not, it returns EOF.

#### See also

`fgetc()`  
`fgets()`  
`fscanf()`

**setbuf()***File I/O***Syntax**

```
#include <stdio.h>

void setbuf(FILE *f, char *buf);
```

**Description**

`setbuf()` lets you specify how a file is buffered. If `buf` is `NULL`, the file is unbuffered; i.e., all input or output goes directly to and comes directly from the file. If `buf` is not `NULL`, it is used as a buffer (`buf` should point to an array of `BUFSIZ` bytes).

**See also**

`fflush()`  
`setvbuf()`

### setjmp()

#### Syntax

```
#include <setjmp.h>

int setjmp(jmp_buf env);
```

#### Description

`setjmp()` saves the current program state in the environment buffer `env` and returns zero. This buffer can be used as a parameter to a later call to `longjmp()`, which then restores the program state and jumps back to the location of the `setjmp`. This time, `setjmp()` returns a non-zero value, which is equal to the second parameter to `longjmp()`.

#### Return

Zero if called directly - non-zero if called by a `longjmp()`.

#### See also

`longjmp()`

**setlocale()***Hardware  
specific***Syntax**

```
#include <locale.h>

char *setlocale(int class, const char *loc);
```

**Description**

`setlocale()` changes the program's locale – either all or just part of it, depending on `class`. The new locale is given by the character string `loc`. The classes allowed are given by Table 16.9.

**Table 16.9 Allowable classes for the `setlocale()` function**

Class	Changes the locale...
LC_ALL	for all classes.
LC_COLLATE	for the <code>strcoll()</code> and <code>strxfrm()</code> functions.
LC_MONETARY	for monetary formatting.
LC_NUMERIC	for numeric formatting.
LC_TIME	for the <code>strftime()</code> function.
LC_TYPE	for character handling and multi-byte character functions.

CodeWarrior supports only the minimum locale “C” (see `locale.h`), so this function has no effect.

**Return**

”C”, if `loc` is “C” or `NULL`; `NULL` otherwise.

**See also**

`localeconv()`  
`strcoll()`  
`strftime()`  
`strxfrm()`

### setvbuf()

File I/O



#### Syntax

```
#include <stdio.h>

void setvbuf(FILE *f,
             char *buf,
             int mode,
             size_t size);
```

#### Description

`setvbuf()` is used to specify how a file is buffered. `mode` determines how the file is buffered.

**Table 16.10** Operating modes for the `setvbuf()` function

Mode	Buffering
<code>_IOFBF</code>	Fully buffered
<code>_IOLBF</code>	Line buffered
<code>_IONBF</code>	Unbuffered

To make a file unbuffered, call `setvbuf()` with mode `_IONBF`; the other arguments (`buf` and `size`) are ignored.

In all other modes, the file uses buffer `buf` of size `size`. If `buf` is `NULL`, the function allocates a buffer of size `size` itself.

#### See also

`fflush()`  
`setbuf()`



## signal()

### Syntax

```
#include <signal.h>

_sig_func signal(int sig, _sig_func handler);
```

### Description

`signal()` defines how the application shall respond to the `sig` signal. The various responses are given by the table below.

**Table 16.11 Various responses to the `signal()` function's input signal**

Handler	Response to the signal
SIG_IGN	The signal is ignored.
SIG_DFL	The default response (HALT).
a function	The function is called with <code>sig</code> as parameter.

The signal handling function is defined as:

```
typedef void (*_sig_func)(int sig);
```

The signal can be raised using the `raise()` function. Before the handler is called, the response is reset to `SIG_DFL`.

In CodeWarrior, there are only two signals: `SIGABRT` indicates an abnormal program termination, and `SIGTERM` a normal program termination.

### Return

If signal succeeds, it returns the previous response for the signal; otherwise it returns `SIG_ERR` and sets `errno` to a positive non-zero value.

### **sin() and sinf()**

#### **Syntax**

```
#include <math.h>

double sin(double x);
float sinf(float x);
```

#### **Description**

`sin()` computes the sine of `x`.

#### **Return**

The sine `sin(x)` of `x` in radians.

#### **See also**

`asin()` and `asinf()`  
`acos()` and `acosf()`  
`atan()` and `atanf()`  
`atan2()` and `atan2f()`  
`cos()` and `cosf()`  
`tan()` and `tanf()`

### **sinh() and sinhf()**

#### **Syntax**

```
#include <math.h>

double sinh(double x);
float  sinhf(float x);
```

#### **Description**

`sinh()` computes the hyperbolic sine of `x`.

#### **Return**

The hyperbolic sine `sinh(x)` of `x`. If it fails because the value is too large, it returns infinity with the same sign as `x` and sets `errno` to `ERANGE`.

#### **See also**

`asin()` and `asinf()`  
`cosh()` and `coshf()`  
`sin()` and `sinf()`  
`tan()` and `tanf()`

### sprintf()

#### Syntax

```
#include <stdio.h>

int sprintf(char *s, const char *format, ...);
```

#### Description

`sprintf()` writes formatted output to string `s`. It evaluates the arguments, converts them according to `format`, and writes the result to `s`, terminated with a zero character.

The format string contains the text to be printed. Any character sequence in `format` starting with '%' is a format specifier that is replaced by the corresponding argument. The first format specifier is replaced with the first argument after `format`, the second format specifier by the second argument, and so on.

A format specifier has the form:

```
FormatSpec      = % {Format}[Width][Precision]
                  [Length]Conversion
```

where:

• `Format` = - | + | <a blank> | # | 0

The format defines justification and sign information (the latter only for numerical arguments). A "-" left justifies the output, a "+" forces output of the sign, and a blank output a blank if the number is positive and a "-" if it is negative. The effect of "#" depends on the conversion character following (Table 16.12).

**Table 16.12 Effect of # in the Format specification**

Conversion	Effect of "#"
e, E, f	The value of the argument always is printed with decimal point, even if there are no fractional digits.
g, G	As above, but In addition zeroes are appended to the fraction until the specified width is reached.
o	A zero is printed before the number to indicate an octal value.

**Table 16.12 Effect of # in the Format specification (*continued*)**

Conversion	Effect of "#"
x, X	"0x" (if the conversion is "x") or "0X" (if it is "X") is printed before the number to indicate a hexadecimal value.
others	undefined.

A "0" as format specifier adds leading zeroes to the number until the desired width is reached, if the conversion character specifies a numerical argument.

If both " " and "+" are given, only "+" is active; if both "0" and "-" are specified, only "-" is active. If there is a precision specification for integral conversions, "0" is ignored.

- `Width` = \* | Number | 0Number

Number defines the minimum field width into which the output is to be put. If the argument is smaller, the space is filled as defined by the format characters.

0Number is the same as above, but 0s are used instead of blanks.

If a "\*" is given, the field width is taken from the next argument, which of course must be a number. If that number is negative, the output is left-justified.

- `Precision` = [Number | \*]

The effect of the precision specification depends on the conversion character.

**Table 16.13 Effect of the Precision specification**

Conversion	Precision
d, i, o, u, x, X	The minimum number of digits to print.
e, E, f	The number of fractional digits to print.
g, G	The maximum number of significant digits to print.
s	The maximum number of characters to print.
others	undefined.

If the precision specifier is "\*", the precision is taken from the next argument, which must be an int. If that value is negative, the precision is ignored.

- `Length` = h | l | L

## The Standard Functions

---

A length specifier tells `sprintf()` what type the argument has. The first two length specifiers can be used in connection with all conversion characters for integral numbers. "h" defines `short`; "l" defines `long`. Specifier "L" is used in conjunction with the conversion characters for floating point numbers and specifies `long double`.

- Conversion = c|d|e|E|f|g|G|i|n|o|p|s|u|x|X|%

These conversion characters have the following meanings:

**Table 16.14 Meaning of the Conversion characters**

Conversion	Description
c	The int argument is converted to unsigned char; the resulting character is printed.
d, i	An int argument is printed.
e, E	The argument must be a double. It is printed in the form [-]d.ddde±dd (scientific notation). The precision determines the number of fractional digits, the digit to the left of the decimal is   0 unless the argument is 0.0. The default precision is 6 digits. If the precision is zero and the format specifier "#" is not given, no decimal point is printed. The exponent always has at least 2 digits; the conversion character is printed just before the exponent.
f	The argument must be a double. It is printed in the form [-]ddd.ddd. See above. If the decimal point is printed, there is at least one digit to the left of it.
g, G	The argument must be a double. <code>sprintf()</code> chooses either format "f" or "e" (or "E" if "G" is given), depending on the magnitude of the value. Scientific notation is used only if the exponent is < -4 or greater than or equal to the precision.
n	The argument must be a pointer to an int. <code>sprintf()</code> writes the number of characters written so far to that address. If "n" is used together with length specifier "h" or "l", the argument must be a pointer to a short int or a long int.
o	The argument, which must be an unsigned int; is printed in octal notation.
p	The argument must be a pointer; its value is printed in hexadecimal notation.

Table 16.14 Meaning of the Conversion characters (*continued*)

Conversion	Description
s	The argument must be a char *; printf() writes the string.
u	The argument, which must be an unsigned int; is written in decimal notation.
x, X	The argument, which must be an unsigned int; is written in hexadecimal notation. "x" uses lower case letters "a" to "f", while "X" uses upper case letters.
%	Prints a "%" sign. Should only be given as "%%".

Conversion characters for integral types are "d", "i", "o", "u", "x", and "X"; for floating point types "e", "E", "f", "g", and "G".

If printf() finds an incorrect format specification, it stops processing, terminates the string with a zero character, and returns successfully.

---

**NOTE** Floating point support increases the printf size considerably, and therefore the define "LIBDEF\_PRINTF\_FLOATING" exists which should be set if no floating point support is used. Some targets contain special libraries without floating point support.

The IEEE64 floating point implementation only supports printing numbers with up to 9 decimal digits. This limitation occurs because the implementation is using unsigned long internally which cannot hold more digits. Supporting more digits would increase the printf size still more and would also cause the application to run considerably slower.

---

### Return

The number of characters written to s.

### See also

sscanf()

### **sqrt() and sqrtf()**

#### **Syntax**

```
#include <math.h>

double sqrt(double x);
float  sqrtf(float x);
```

#### **Description**

`sqrt()` computes the square root of `x`.

#### **Return**

The square root of `x`. If `x` is negative, it returns 0 and sets `errno` to `EDOM`.

#### **See also**

`pow()` and `powf()`



### **srand()**

#### **Syntax**

```
#include <stdlib.h>

void srand(unsigned int seed)
;
```

#### **Description**

`srand()` initializes the seed of the random number generator. The default seed is 1.

#### **See also**

`rand()`

### sscanf()

#### Syntax

```
#include <stdio.h>

int sscanf(const char *s, const char *format, ...);
```

#### Description

`sscanf()` scans string `s` according to the given format, storing the values in the given parameters. The format specifiers in the format tell `sscanf()` what to expect next. A format specifier has the format:

• `FormatSpec = %[Flag][Width][Size]Conversion`

where:

- `Flag = *`

If the "\*" sign which starts a format specification is followed by a "\*", the scanned value is not assigned to the corresponding parameter.

- `Width = Number`

Specifies the maximum number of characters to read when scanning the value. Scanning also stops if white space or a character not matching the expected syntax is reached.

- `Size = h|l|L`

Specifies the size of the argument to read. The meaning is given in Table 16.15.

**Table 16.15 Relationship of the Size parameter with allowable conversions and types.**

Size	Allowable Conversions	Parameter Type
h	d, i, n	short int * (instead of int *)
h	o, u, x, X	unsigned short int * (instead of unsigned int *)
l	d, i, n	long int * (instead of int *)
l	o, u, x, X	unsigned long int * (instead of unsigned int *)

**Table 16.15 Relationship of the Size parameter with allowable conversions and types.**

Size	Allowable Conversions	Parameter Type
l	e, E, f, g, G	double * (instead of float *)
L	e, E, f, g, G	long double * (instead of float *)

- Conversion = c|d|e|E|f|g|G|i|n|o|p|s|u|x|X|%|Range

These conversion characters tell `sscanf()` what to read and how to store it in a parameter. Their meaning is shown in Table 16.16.

## The Standard Functions

---

**Table 16.16** Description of the action taken for each conversion.

Conversion	Description
c	Reads a string of exactly <code>width</code> characters and stores it in the parameter. If no <code>width</code> is given, one character is read. The argument must be a <code>char *</code> . The string read is <i>not</i> zero-terminated.
d	A decimal number (syntax below) is read and stored in the parameter. The parameter must be a pointer to an integral type.
i	As "d", but also reads octal and hexadecimal numbers (syntax below).
e, E, f, g, or G	Reads a floating point number (syntax below). The parameter must be a pointer to a floating-point type.
n	The argument must be a pointer to an <code>int</code> . <code>sscanf()</code> writes the number of characters read so far to that address. If "n" is used together with length specifier "h" or "l", the argument must be a pointer to a <code>short int</code> or a <code>long int</code> .
o	Reads an octal number (syntax below). The parameter must be a pointer to an integral type.
p	Reads a pointer in the same format as <code>sprintf()</code> prints it. The parameter must be a <code>void **</code> .
s	Reads a character string up to the next white space character or at most <code>width</code> characters. The string is zero-terminated. The argument must be of type <code>char *</code> .
u	As "d", but the parameter must be a pointer to an unsigned integral type.
x, X	As "u", but reads a hexadecimal number.
%	Skips a "%" sign in the input. Should only be given as "%%".

Range = `[^List]`

List = `Element {Element}`

Element = `<any char> [-<any char>]`

You can also use a scan set to read a character string that either contains only the given characters or contains only characters not in the set. A scan set always is

bracketed by left and right brackets. If the first character in the set is "<sup>^</sup>", the set is inverted (i.e., only characters *not* in the set are allowed). You can specify whole character ranges, e.g., "A-Z" specifies all upper-case letters. If you want to include a right bracket in the scan set, it must be the first element in the list, a dash (" - ") must be either the first or the last element. A "<sup>^</sup>" that shall be included in the list instead of indicating an inverted list must not be the first character after the left bracket.

Some examples are:

```
[A-Za-z]  Allows all upper- and lower-case characters.
[^A-Z]    Allows any character that is not an upper-case character.
[]abc]    Allows "]", "a", "b" and "c".
[^]abc]   Allows any char except "]", "a", "b" and "c".
[-abc]    Allows "-", "a", "b" and "c".
```

A white space in the format string skips all white space characters up to the next non-white-space character. Any other character in the format must be exactly matched by the input; otherwise `sscanf()` stops scanning.

The syntax for numbers as scanned by `sscanf()` is the following:

---

```
Number      = FloatNumber | IntNumber
IntNumber   = DecNumber | OctNumber | HexNumber
DecNumber   = SignDigit {Digit}
OctNumber   = Sign0 {OctDigit}
HexNumber   = 0 (x|X) HexDigit {HexDigit}
FloatNumber = Sign {Digit} [.{Digit}] [Exponent]
Exponent    = (e|E) DecNumber
OctDigit    = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Digit       = OctDigit | 8 | 9
HexDigit    = Digit | A | B | C | D | E | F | a | b | c | d | e | f
```

---

### Return

EOF, if `s` is NULL; otherwise it returns the number of arguments filled in.

---

**NOTE** If `sscanf()` finds an illegal input (i.e., not matching the required syntax), it simply stops scanning and returns successfully!

---

### **strcat()**

#### **Syntax**

```
#include <string.h>

char *strcat(char *p, const char *q);
```

#### **Description**

`strcat()` appends string `q` to the end of string `p`. Both strings and the resulting concatenation are zero-terminated.

#### **Return**

`p`

#### **See also**

`memcpy()` and `memmove()`  
`strcpy()`  
`strncat()`  
`strncpy()`

### **strchr()**

#### **Syntax**

```
#include <string.h>

char *strchr(const char *p, int ch);
```

#### **Description**

`strchr()` looks for character `ch` in string `p`. If `ch` is `'\0'`, the function looks for the end of the string.

#### **Return**

A pointer to the character, if found; if there is no such character in `*p`, `NULL` is returned.

#### **See also**

```
memchr()
strchr()
strstr()
```

### strcmp()

#### Syntax

```
#include <string.h>

int strcmp(const char *p, const char *q);
```

#### Description

`strcmp()` compares the two strings, using the character ordering given by the ASCII character set.

#### Return

A negative integer, if `p` is smaller than `q`; zero, if both strings are equal; or a positive integer if `p` is greater than `q`.

---

**NOTE** The return value of `strcmp()` is such that it could be used as a comparison function in `bsearch()` and `qsort()`.

---

#### See also

```
memcmp()
strcoll()
strncmp()
```



### **strcoll()**

#### **Syntax**

```
#include <string.h>

int strcoll(const char *p, const char *q);
```

#### **Description**

`strcoll()` compares the two strings interpreting them according to the current locale, using the character ordering given by the ASCII character set.

#### **Return**

A negative integer, if `p` is smaller than `q`; zero, if both strings are equal; or a positive integer if `p` is greater than `q`.

#### **See also**

```
memcmp()
strcpy()
strncmp()
```

### **strcpy()**

#### **Syntax**

```
#include <string.h>

char *strcpy(char *p, const char *q);
```

#### **Description**

`strcpy()` copies string `q` into string `p` (including the terminating `'\0'`).

#### **Return**

`p`

#### **See also**

`memcpy()` and `memmove()`  
`strncpy()`

### **strcspn()**

#### **Syntax**

```
#include <string.h>

size_t strcspn(const char *p, const char *q);
```

#### **Description**

`strcspn()` searches `p` for the first character that also appears in `q`.

#### **Return**

The length of the initial segment of `p` that contains only characters *not* in `q`.

#### **See also**

```
strchr()
strpbrk()
strrchr()
strspn()
```

### **strerror()**

#### **Syntax**

```
#include <string.h>

char *strerror(int errno);
```

#### **Description**

`strerror()` returns an error message appropriate for error number `errno`.

#### **Return**

A pointer to the message string.

#### **See also**

`perror()`

## strftime()

### Syntax

```
#include <time.h>

size_t strftime(char *s,
                size_t max,
                const char *format,
                const struct tm *time);
```

### Description

`strftime()` converts `time` to a character string `s`. If the conversion results in a string longer than `max` characters (including the terminating `'\0'`), `s` is left unchanged and the function returns unsuccessfully. How the conversion is done is determined by the `format` string. This string contains text, which is copied one-to-one to `s`, and format specifiers. The latter always start with a `'%'` sign and are replaced by the following:

**Table 16.17** `strftime()` output string content and format

Format	Replaced with
<code>%a</code>	Abbreviated name of the weekday of the current locale, e.g., "Fri".
<code>%A</code>	Full name of the weekday of the current locale, e.g., "Friday".
<code>%b</code>	Abbreviated name of the month of the current locale, e.g., "Feb".
<code>%B</code>	Full name of the month of the current locale, e.g., "February".
<code>%c</code>	Date and time in the form given by the current locale.
<code>%d</code>	Day of the month in the range from 0 to 31.
<code>%H</code>	Hour, in 24-hour-clock format.
<code>%I</code>	Hour, in 12-hour-clock format.
<code>%j</code>	Day of the year, in the range from 0 to 366.
<code>%m</code>	Month, as a decimal number from 0 to 12.
<code>%M</code>	Minutes

## The Standard Functions

---

**Table 16.17** `strftime()` output string content and format (*continued*)

Format	Replaced with
<code>%p</code>	AM/PM specification of a 12-hour clock or equivalent of current locale.
<code>%S</code>	Seconds
<code>%U</code>	Week number in the range from 0 to 53, with Sunday as the first day of the first week.
<code>%w</code>	Day of the week (Sunday = 0, Saturday = 6).
<code>%W</code>	Week number in the range from 0 to 53, with Monday as the first day of the first week.
<code>%x</code>	The date in format given by current locale.
<code>%X</code>	The time in format given by current locale.
<code>%y</code>	The year in short format, e.g., "93".
<code>%Y</code>	The year, including the century (e.g., "1993").
<code>%Z</code>	The time zone, if it can be determined.
<code>%%</code>	A single '%' sign.

### Return

If the resulting string would have had more than `max` characters, zero is returned; otherwise the length of the created string is returned.

### See also

`mktime()`  
`setlocale()`  
`time()`

### **strlen()**

#### **Syntax**

```
#include <string.h>

size_t strlen(const char *s);
```

#### **Description**

`strlen()` returns the number of characters in string `s`.

#### **Return**

The length of the string.

### strncat()

#### Syntax

```
#include <string.h>
```

```
char *strncat(char *p, const char *q, size_t n);
```

#### Description

`strncat()` appends string `q` to string `p`. If `q` contains more than `n` characters, only the first `n` characters of `q` are appended to `p`. The two strings and the result all are zero-terminated.

#### Return

`p`

#### See also

`strcat()`



### **strncmp()**

#### **Syntax**

```
#include <string.h>
```

```
char *strncmp(char *p, const char *q, size_t n);
```

#### **Description**

`strncmp()` compares at most the first `n` characters of the two strings.

#### **Return**

A negative integer, if `p` is smaller than `q`; zero, if both strings are equal; or a positive integer if `p` is greater than `q`.

#### **See also**

`memcmp()`

`strcmp()`

### strncpy()

#### Syntax

```
#include <string.h>

char *strncpy(char *p, const char *q, size_t n);
```

#### Description

`strncpy()` copies at most the first `n` characters of string `q` to string `p`, overwriting `p`'s previous contents. If `q` contains less than `n` characters, a `'\0'` is appended.

#### Return

`p`

#### See also

`memcpy()`  
`strcpy()`

### **strpbrk()**

#### **Syntax**

```
#include <string.h>

char *strpbrk(const char *p, const char *q);
```

#### **Description**

`strpbrk()` searches for the first character in `p` that also appears in `q`.

#### **Return**

NULL, if there is no such character in `p`; a pointer to the character otherwise.

#### **See also**

```
strchr()
strcspn()
strrchr()
strspn()
```

### **strrchr()**

#### **Syntax**

```
#include <string.h>

char *strrchr(const char *s, int c);
```

#### **Description**

`strpbrk()` searches for the last occurrence of character `ch` in `s`.

#### **Return**

`NULL`, if there is no such character in `p`; a pointer to the character otherwise.

#### **See also**

```
strchr()
strcspn()
strpbrk()
strspn()
```

### **strspn()**

#### **Syntax**

```
#include <string.h>

size_t strspn(const char *p, const char *q);
```

#### **Description**

`strspn()` returns the length of the initial part of `p` that contains only characters also appearing in `q`.

#### **Return**

The position of the first character in `p` that is not in `q`.

#### **See also**

```
strchr()
strcspn()
strpbrk()
strrchr()
```

### **strstr()**

#### **Syntax**

```
#include <string.h>
```

```
char *strstr(const char *p, const char *q);
```

#### **Description**

`strstr()` looks for substring `q` appearing in string `p`.

#### **Return**

A pointer to the beginning of the first occurrence of string `q` in `p`, or `NULL`, if `q` does not appear in `p`.

#### **See also**

`strchr()`

`strcspn()`

`strpbrk()`

`strrchr()`

`strspn()`

## strtod()

### Syntax

```
#include <stdlib.h>

double strtod(const char *s, char **end);
```

### Description

`strtod()` converts string `s` into a floating point number, skipping over any white space at the beginning of `s`. It stops scanning when it reaches a character not matching the required syntax and returns a pointer to that character in `*end`. The number format `strtod()` accepts is:

```
FloatNum = Sign{Digit}[.{Digit}][Exp]
Sign      = [+|-]
Exp       = (e|E)SignDigit{Digit}
Digit     = <any decimal digit from 0 to 9>
```

### Return

The floating point number read. If an underflow occurred, `0.0` is returned. If the value causes an overflow, `HUGE_VAL` is returned. In both cases, `errno` is set to `ERANGE`.

### See also

```
atof()
scanf()
strtol()
strtoul()
```

### strtok()

#### Syntax

```
#include <string.h>

char *strtok(char *p, const char *q);
```

#### Description

`strtok()` breaks the string `p` into tokens which are separated by at least one character appearing in `q`. The first time, call `strtok()` using the original string as the first parameter. Afterwards, pass `NULL` as first parameter: `strtok()` will continue at the position it stopped the previous time. `strtok()` saves the string `p` if it isn't `NULL`.

---

**NOTE** This function is not re-entrant because it uses a global variable for saving string `p`. ANSI defines this function in this way.

---

#### Return

A pointer to the token found, or `NULL`, if no token was found.

#### See also

```
strchr()
strcspn()
strpbrk()
strrchr()
strspn()
strstr()
```



## strtol()

### Syntax

```
#include <stdlib.h>

long strtol(const char *s, char **end, int base);
```

### Description

`strtol()` converts string `s` into a long `int` of base `base`, skipping over any white space at the beginning of `s`. It stops scanning when it reaches a character not matching the required syntax (or a character too large for a given base) and returns a pointer to that character in `*end`.

The number format `strtol()` accepts is:

---

```
Int_Number   = Dec_Number|Oct_Number|Hex_Number|Other_Number
Dec_Number   = SignDigit{Digit}
Oct_Number   = Sign0{OctDigit}
Hex_Number   = 0(x|X)Hex_Digit{Hex_Digit}
Other_Number = SignOther_Digit{Other_Digit}
Oct_Digit    = 0|1|2|3|4|5|6|7
Digit        = Oct_Digit|8|9
Hex_Digit    = Digit |A|B|C|D|E|F|a|b|c|d|e|f
Other_Digit  = Hex_Digit|<any char between 'G' and 'Z'|>|<any char between 'g' and 'z'|>
```

---

The base must be 0 or in the range from 2 to 36. If it is between 2 and 36, `strtol()` converts a number in that base (digits larger than 9 are represented by upper or lower case characters from 'A' to 'Z'). If base is zero, the function uses the prefix to find the base. If the prefix is "0", base 8 (octal) is assumed. If it is "0x" or "0X", base 16 (hexadecimal) is taken. Any other prefixes make `strtol()` scan a decimal number.

### Return

The number read. If no number is found, zero is returned; if the value is smaller than `LONG_MIN` or larger than `LONG_MAX`, `LONG_MIN` or `LONG_MAX` is returned and `errno` is set to `ERANGE`.

## The Standard Functions

---

### See also

`atoi()`  
`atol()`  
`scanf()`  
`strtod()`  
`strtoul()`

### strtoul()

#### Syntax

```
#include <stdlib.h>

unsigned long strtoul(const char *s,
                    char **end,
                    int base);
```

#### Description

`strtoul()` converts string `s` into an unsigned long int of base `base`, skipping over any white space at the beginning of `s`. It stops scanning when it reaches a character not matching the required syntax (or a character too large for a given base) and returns a pointer to that character in `*end`. The number format `strtoul()` accepts is the same as for `strtol()` except that the negative sign is not allowed, and so are the possible values for `base`.

#### Return

The number read. If no number is found, zero is returned; if the value is larger than `ULONG_MAX`, `ULONG_MAX` is returned and `errno` is set to `ERANGE`.

#### See also

```
atoi()
atol()
scanf()
strtod()
strtol()
```

### **strxfrm()**

#### **Syntax**

```
#include <string.h>

size_t strxfrm(char *p, const char *q, size_t n);
```

#### **Description**

`strxfrm()` transforms string `q` according to the current locale, such that the comparison of two strings converted with `strxfrm()` using `strcmp()` yields the same result as a comparison using `strcoll()`. If the resulting string would be longer than `n` characters, `p` is left unchanged.

#### **Return**

The length of the converted string.

#### **See also**

```
setlocale()
strcmp()
strcoll()
```

### **system()**

*Hardware  
specific*



#### **Syntax**

```
#include <string.h>

int system(const char *cmd);
```

#### **Description**

`system()` executes the command line `cmd`.

#### **Return**

Zero

### tan() and tanf()

#### Syntax

```
#include <math.h>

double tan(double x);
float tanf(float x);
```

#### Description

`tan()` computes the tangent of  $x$ .  $x$  should be in radians.

#### Return

`tan(x)`. If  $x$  is an odd multiple of  $\pi/2$ , it returns infinity and sets `errno` to `EDOM`.

#### See also

`acos()` and `acosf()`  
`asin()` and `asinf()`  
`atan()` and `atanf()`  
`atan2()` and `atan2f()`  
`cosh()` and `coshf()`  
`sin()` and `sinf()`  
`tan()` and `tanf()`

## **tanh() and tanhf()**

### **Syntax**

```
#include <math.h>

double tanh(double x);
float tanhf(float x);
```

### **Description**

`tanh()` computes the hyperbolic tangent of `x`.

### **Return**

`tanh(x)`

### **See also**

`atan()` and `atanf()`  
`atan2()` and `atan2f()`  
`cosh()` and `coshf()`  
`sin()` and `sinf()`  
`tan()` and `tanf()`

### **time()**

*Hardware  
specific*



#### **Syntax**

```
#include <time.h>

time_t time(time_t *timer);
```

#### **Description**

`time()` gets the current calendar time. If `timer` is not `NULL`, it is assigned to it.

#### **Return**

The current calendar time.

#### **See also**

```
clock()
mktime()
strptime()
```



### tmpfile()

*File I/O*



#### Syntax

```
#include <stdio.h>

FILE *tmpfile(void);
```

#### Description

`tmpfile()` creates a new temporary file using mode "wb+". Temporary files automatically are deleted when they are closed or the application ends.

#### Return

A pointer to the file descriptor if the file could be created; `NULL` otherwise.

#### See also

`fopen()`  
`tmpnam()`

### tmpnam()

*File I/O*



#### Syntax

```
#include <stdio.h>

char *tmpnam(char *s);
```

#### Description

`tmpnam()` creates a new unique filename. If `s` is not `NULL`, this name is assigned to it.

#### Return

A unique filename

#### See also

`tmpfile()`

### **tolower()**

#### **Syntax**

```
#include <ctype.h>

int tolower(int ch);
```

#### **Description**

`tolower()` converts any upper-case character in the range from 'A' to 'Z' into a lower-case character from 'a' to 'z'.

#### **Return**

If `ch` is an upper-case character, the corresponding lower-case letter. Otherwise, `ch` is returned (unchanged).

#### **See also**

`islower()`  
`isupper()`  
`toupper()`

### toupper()

#### Syntax

```
#include <ctype.h>

int toupper(int ch);
```

#### Description

`toupper()` converts any lower-case character in the range from 'a' to 'z' into an upper-case character from 'A' to 'Z'.

#### Return

If `ch` is a lower-case character, the corresponding upper-case letter. Otherwise, `ch` is returned (unchanged).

#### See also

`islower()`  
`isupper()`  
`tolower()`

### ungetc()

*File I/O*



#### Syntax

```
#include <stdio.h>

int ungetc(int ch, FILE *f)
;
```

#### Description

`ungetc()` pushes the single character `ch` back onto the input stream `f`. The next read from `f` will read that character.

#### Return

`ch`

#### See also

`fgets()`  
`fopen()`  
`getc()`  
`getchar()`

## The Standard Functions

---

---

### **va\_arg(), va\_end(), and va\_start()**

#### **Syntax**

```
#include <stdarg.h>

void va_start(va_list args, param);
type va_arg(va_list args, type);
void va_end(va_list args);
```

#### **Description**

These macros can be used to get the parameters in an open parameter list. Calls to `va_arg()` get a parameter of the given type. The following example shows how to do it:

---

```
void my_func(char *s, ...) {
    va_list args;
    int     i;
    char    *q;

    va_start(args, s);
    /* First call to 'va_arg' gets the first arg. */
    i = va_arg (args, int);
    /* Second call gets the second argument. */
    q = va_arg(args, char *);
    ...
    va_end (args);
}
```

---

---

**vfprintf(), vprintf(), and vsprintf()**

File I/O

**Syntax**

---

```
#include <stdio.h>

int vfprintf(FILE *f,
             const char *format,
             va_list args);
int vprintf(const char *format, va_list args);
int vsprintf(char *s,
             const char *format,
             va_list args);
```

---

**Description**

These functions are the same as `fprintf()`, `printf()`, and `sprintf()`, except that they take a `va_list` instead of an open parameter list as argument.

For a detailed format description see `sprintf()`.

---

**NOTE** Only `vsprintf()` is implemented, because the other two functions depend on the actual setup and environment of the target.

---

**Return**

The number of characters written, if successful; a negative number otherwise.

**See also**

`va_arg()`, `va_end()`, and `va_start()`

### wctomb()

#### Syntax

```
#include <stdlib.h>

int wctomb(char *s, wchar_t wchar);
```

#### Description

`wctomb()` converts `wchar` to a multi-byte character, stores that character in `s`, and returns the length in bytes of `s`.

#### Return

The length of `s` in bytes after the conversion.

#### See also

`wctombs()`



### **wcstombs()**

*Hardware  
specific*



#### **Syntax**

```
#include <stdlib.h>

int wcstombs(char *s, const wchar_t *ws, size_t n);
```

#### **Description**

`wcstombs()` converts the first `n` wide character codes in `ws` to multi-byte characters, stores them character in `s`, and returns the number of wide characters converted.

#### **Return**

The number of wide characters converted.

#### **See also**

`wctomb()`

## The Standard Functions

---

# Appendices

---

The appendices covered in this manual are:

- Porting Tips and FAQs: Hints about EBNF notation used by the linker and about porting applications from other Compiler vendors to this Compiler
- Global Configuration-File Entries: Documentation for the entries in the mcutools.ini file
- Local Configuration-File Entries: Documentation for the entries in the project.ini file.



# Porting Tips and FAQs

---

This appendix describes some FAQs and provides tips on the syntax of EBNF or how to port the application from a different tool vendor.

## Migration Hints

This section describes the differences between this compiler and the compilers of other vendors. It also provides information about porting sources and how to adapt them.

### Porting from Cosmic

If your current application is written for Cosmic compilers, there are some special things to consider.

#### How to Get Started...

The best way is if you create a new project using the New Project Wizard (in the CodeWarrior IDE Menu File > New) or a project from a stationery template. This will set up a project for you with all the default options and library files included. Then add the existing files used for Cosmic to the project (e.g., through drag & drop from the Windows Explorer or using in the CodeWarrior IDE: the menu Project > Add Files. Make sure that the right memory model and CPU type are used as for the Cosmic project.

### Cosmic Compatibility Mode Switch

The latest compiler offers a Cosmic compatibility mode switch (-Ccx: Cosmic Compatibility Mode for Space Modifiers and Interrupt Handlers). Enable this compiler option so the compiler accepts most Cosmic constructs.

### Assembly Equates

For the Cosmic compiler, you need to define equates for the inline assembly using `equ`. If you want to use an equate or value in C as well, you need to define it using `#define` as well. For this compiler, you only need one version (i.e., use `#define`) both for C and for inline assembly (Listing A.1). The `equ` directive is not supported in normal C code.

## Porting Tips and FAQs

### Migration Hints

---

#### Listing A.1 An example using the EQU directive

---

```
#ifdef __MWERKS__
#define CLKSRC_B 0x00 /*; Clock source */
#else
CLKSRC_B : equ $00 ; Clock source
#endif
```

---

## Inline Assembly Identifiers

For the Cosmic compiler, you need to place an underscore (‘\_’) in front of each identifier, but for this compiler you can use the same name both for C and inline assembly. In addition, for better type-safety with this compiler you need to place a ‘@’ in front of variables if you want to use the address of a variable. Using a conditional block like the one below in Listing A.2.

#### Listing A.2 Using a conditional block to account for different compilers

---

```
#ifdef __MWERKS__
ldx @myVariable,x
jsr MyFunction
#else
ldx _myVariable,x
jsr _MyFunction
#endif
```

---

Using macros which deal with the cases below (Listing A.3) is a better way to deal with this.

#### Listing A.3 Using a macro to account for different compilers

---

```
#ifdef __MWERKS__
#define USCR(ident) ident
#define USCRA(ident) @ ident
#else /* for COSMIC, add a _ (underscore) to each ident */
#define USCR(ident) _##ident
#define USCRA(ident) _##ident
#endif
```

---

so the source can use the macros:

```
ldx USCRA(myVariable),x  
jsr USCR(MyFunction)
```

## Pragma Sections

Cosmic uses the `#pragma` section syntax, while this compiler employs either `#pragma DATA_SEG` (Listing A.4) or `#pragma CONST_SEG` (Listing A.5).

or another example (for the data section):

### Listing A.4 #pragma DATA\_SEG

---

```
#ifdef __MWERKS__  
#pragma DATA_SEG APPLDATA_SEG  
#else  
#pragma section {APPLDATA}  
#endif
```

---

### Listing A.5 #pragma CONST\_SEG

---

```
#ifdef __MWERKS__  
#pragma CONST_SEG CONSTVECT_SEG  
#else  
#pragma section const {CONSTVECT}  
#endif
```

---

Do not forget to use the segments (in the examples above `CONSTVECT_SEG` and `APPLDATA_SEG`) in the linker `*.prm` file in the `PLACEMENT` block.

## Inline Assembly Constants

Cosmic uses an assembly constant syntax, whereas this compiler employs the normal C constant syntax (Listing A.6):

### Listing A.6 Normal C constant syntax

---

```
#ifdef __MWERKS__  
    and 0xF8  
#else  
    and #$F8
```

```
#endif
```

---

## Inline Assembly and Index Calculation

Cosmic uses the + operator to calculate offsets into arrays. For CodeWarrior, you have to use a colon (:): instead:

### Listing A.7 Using a colon for offset

---

```
ldx array:7
#else
    ldx array+7
#endif
```

---

## Inline Assembly and Tabs

Cosmic lets you use TAB characters in normal C strings (surrounded by double quotes):

```
asm("This string contains hidden tabs!");
```

Because the compiler rejects hidden tab characters in C strings according to the ANSI-C standard, you need to remove the tab characters from such strings.

## Inline Assembly and Operators

Cosmic's and this compiler's inline assembly may not support the same amount or level of operators. But in most cases it is simple to rewrite or transform them (Listing A.8)

### Listing A.8 Accounting for different operators among different compilers

---

```
#ifdef __MWERKS__
    ldx #(BOFFIE + WUPIE) ; enable Interrupts
#else
    ldx #(BOFFIE | WUPIE) ; enable Interrupts
#endif
#ifdef __MWERKS__
    lda  #(_TxBuf2+Data0)
    ldx  #((_TxBuf2+Data0) / 256)
#else
    lda  #((_TxBuf2+Data0) & $ff)
    ldx  #(((_TxBuf2+Data0) >> 8) & $ff)
```

---



```
#endif
```

---

## @interrupt

Cosmic uses the @interrupt syntax, whereas this compiler employs the interrupt syntax. In order to keep the source base portable, a macro can be used (e.g., in a main header file which selects the correct syntax depending on the compiler used:

### Listing A.9 interrupt syntax

---

```
/* place the following in a header file: */
#ifdef __MWERKS__
    #define INTERRUPT interrupt
#else
    #define INTERRUPT @interrupt
#endif
```

---

```
/* now for each @interrupt we use the INTERRUPT macro: */
void INTERRUPT myISRFunction(void) { ....
```

## Inline Assembly and Conditional Blocks

In most cases, the (-Ccx: Cosmic Compatibility Mode for Space Modifiers and Interrupt Handlers) will handle the #asm blocks used in Cosmic inline assembly code Cosmic compatibility switch. However, if #asm is used with conditional blocks like #ifdef or #if, then the C parser may not accept it (Listing A.10).

### Listing A.10 Use of conditional blocks without asm block markers ( { and })

---

```
void foo(void) {
    #asm
        nop
    #if 1
        #endasm
        foo();
        #asm
            #endif
            nop
        #endasm
    }
```

---

## Porting Tips and FAQs

### Migration Hints

---

In such case, the `#asm` and `#endasm` must be ported to `asm { and }` block markers (Listing A.11)

#### Listing A.11 Use of Conditional Blocks with `asm { and }` Block Markers

---

```
void foo(void) {
    asm { // asm #1
        nop
    }
    #if 1
        } // end of asm #1
        foo();
        asm { // asm #2
    #endif
        nop
    } // end of asm #2
}
```

---

## Compiler Warnings

Check carefully the warnings produced by the compiler. The Cosmic compiler does not warn about many cases where your application code may contain a bug. Later on the warnings can be switched off if they are OK (e.g., using the `-W2: No Information and Warning Messages` option or using `#pragma MESSAGE: Message Setting` in the source code).

## Linker \*.lcf File (for the Cosmic compiler) and Linker \*.prm File (for the Freescale Compiler)

Cosmic uses a `*.lcf` file for the linker with a special syntax. This compiler uses a linker parameter file with a `*.prm` file extension. The syntax is not the same format, but most things are straightforward to port. For this compiler, you must declare the RAM or ROM areas in the `SEGMENTS . . . END` block and place the sections into the `SEGMENTS` in the `PLACEMENT . . . END` block.

Make sure that all your segments you declared in your application (through `#pragma DATA_SEG`, `#pragma CONST_SEG`, and `#pragma CODE_SEG`) are used in the `PLACEMENT` block of the Linker prm file.

Check the linker warnings or errors carefully. They may indicate what you need to adjust or correct in your application. E.g., you may have allocated the vectors in the linker `*.prm` file (using `VECTOR` or `ADDRESS` syntax) and allocated them as well in the application itself (e.g., with the `#pragma CONST_SEG` or with the `@address` syntax). Allocating objects twice is an error, so these objects must be allocated one or the other way, but not both.

Consult your map file produced by the linker to check that everything is correctly allocated.

Remember that the linker is a smart linker. This means that objects not used or referenced are not linked to the application. The Cosmic linker may link objects even if they are not used or referenced, but, nevertheless, these objects may still be required to be linked to the application for some reason not required by the linker. In order to have objects linked to the application regardless if they are used or not, use the `ENTRIES ... END` block in the Linker `*.prm` file:

```
ENTRIES /* the following objects or variables need to be
linked even if not referenced by the application */
```

```
_vectab ApplHeader FlashEraseTable
END
```

## Allocation of Bitfields

Allocation of bitfields is very compiler-dependent. Some compilers allocate the bits first from right (LSByte) to left (MSByte), and others allocate from left to right. Also, alignment and byte or word crossing of bitfields is not implemented consistently. Some possibilities are to:

- Check the different allocation strategies,
- Check if there is an option to change the allocation strategy in the compiler, or
- Use the compiler defines to hold sources portable:

```
- __BITFIELD_LSBIT_FIRST__
- __BITFIELD_MSBIT_FIRST__
- __BITFIELD_LSBYTE_FIRST__
- __BITFIELD_MSBYTE_FIRST__
- __BITFIELD_LSWORD_FIRST__
- __BITFIELD_MSWORD_FIRST__
- __BITFIELD_TYPE_SIZE_REDUCTION__
- __BITFIELD_NO_TYPE_SIZE_REDUCTION__
```

## Type Sizes and Sign of char

Carefully check the type sizes that a particular compiler uses. Some compilers implement the sizes for the standard types (`char`, `short`, `int`, `long`, `float`, or `double`) differently. For instance, the size for an `int` is 16 bits for some compilers and 32 bits for others.

If necessary, change the default type settings with the `-T` option.

The sign of `plain char` is also not consistent for all compilers. If the software program requires that `char` be signed or unsigned, either change all `plain char` types to the signed or unsigned types or change the sign of `char` with the `-T`: Flexible Type Management option.

## @bool Qualifier

Some compiler vendors provide a special keyword `@bool` to specify that a function returns a boolean value:

```
@bool int foo(void);
```

Because this special keyword is not supported, remove `@bool` or use a define such as this:

```
#define _BOOL /*@bool*/  
_BOOL int foo(void);
```

## @tiny and @far Qualifier for Variables

Some compiler vendors provide special keywords to place variables in absolute locations. Such absolute locations can be expressed in ANSI-C as constant pointers:

```
#ifdef __HIWARE__  
#define REG_PTB (*(volatile char*)(0x01))  
#else /* other compiler vendors use non-ANSI features */  
@tiny volatile char REG_PTB @0x01; /* port B */  
#endif
```

The Compiler does not need the `@tiny` qualifier directly. The Compiler is smart enough to take the right addressing mode depending on the address:

```
/* compiler uses the correct addressing mode */  
volatile char REG_PTB @0x01;
```

## Arrays with Unknown Size

Some compilers accept the following non-ANSI compliant statement to declare an array with an unknown size:

```
extern char buf[0];
```

However, the compiler will issue an error message for this because an object with size zero (even if declared as extern) is illegal. Use the legal version:

```
extern char buf[];
```

## Missing Prototype

Many compilers accept a function-call usage without a prototype. This compiler will issue a warning for this. However if the prototype of a function with open arguments is missing or this function is called with a different number of arguments, this is clearly an error:

```
printf("hello world!"); // compiler assumes void
printf(char*);
// error, argument number mismatch!
printf("hello %s!", "world");
```

To avoid such programming bugs use the `-Wpd`: Error for Implicit Parameter Declaration compiler option and always include or provide a prototype.

## `_asm("sequence")`

Some compilers use `_asm("string")` to write inline assembly code in normal C source code: `_asm("nop");`

This can be rewritten with `asm` or `asm {}: asm nop;`

## Recursive Comments

Some compilers accept recursive comments without any warnings. The Compiler will issue a warning for each such recursive comment:

```
/* this is a recursive comment */
   int a;
/* */
```

The Compiler will treat the above source completely as one single comment, so the definition of ‘a’ is inside the comment. That is, the Compiler treats everything between the first opening comment ‘/\*’ until the closing comment token ‘\*/’ as a comment. If there are such recursive comments, correct them.

## Interrupt Function, @interrupt

Interrupt functions have to be marked with `#pragma TRAP_PROC` or using the `interrupt` keyword (Listing A.12).

### Listing A.12 Using the TRAP\_PROC pragma with an Interrupt Function

---

```
#ifdef __HIWARE__
    #pragma TRAP_PROC
    void MyTrapProc(void)
#else /* other compiler-vendor non-ANSI declaration of interrupt
      function */
    @interrupt void MyTrapProc(void)
#endif
{
    /* code follows here */
}
```

---

## Defining Interrupt Functions

This manual section discusses some important topics related to the handling of interrupt functions:

- Definition of an interrupt function
- Initialization of the vector table
- Placing an interrupt function in a special section

## Defining an Interrupt Function

The compiler provides two ways to define an interrupt function:

- Using `pragma TRAP_PROC`.
- Using the keyword `interrupt`.

## Using the “TRAP\_PROC” Pragma

The `TRAP_PROC` pragma informs the compiler that the following function is an interrupt function (Listing A.13). In that case, the compiler should terminate the function by a special interrupt return sequence (for many processors, an RTI instead of an RTS).

### Listing A.13 Example of using the TRAP\_PROC pragma

---

```
#pragma TRAP_PROC
void INCcount(void) {
```

---

```
tcount++;  
}
```

---

## Using the “interrupt” keyword

The “interrupt” keyword is non-standard ANSI-C and therefore is not supported by all ANSI-C compiler vendors. In the same way, the syntax for the usage of this keyword may change between different compilers. The keyword `interrupt` informs the compiler that the following function is an interrupt function (Listing A.14).

### Listing A.14 Example of using the “interrupt” keyword

---

```
interrupt void INCcount(void) {  
    tcount++;  
}
```

---

## Initializing the Vector Table

Once the code for an interrupt function has been written, you must associated this function with an interrupt vector. This is done through initialization of the vector table. You can initialize the vector table in the following ways:

- Using the `VECTOR ADDRESS` or `VECTOR` command in the PRM file
- Using the “interrupt” keyword.

## Using the Linker Commands

The Linker provides two commands to initialize the vector table: `VECTOR ADDRESS` or `VECTOR`. You use the `VECTOR ADDRESS` command to write the address of a function at a specific address in the vector table.

In order to enter the address of the `INCcount()` function at address `0x8A`, insert the following command in the application’s PRM file (Listing A.15).

### Listing A.15 Using the `VECTOR ADDRESS` command

---

```
VECTOR ADDRESS 0x8A INCcount
```

---

The `VECTOR` command is used to associate a function with a specific vector, identified with its number. The mapping from the vector number is target-specific.

---

## Porting Tips and FAQs

### Migration Hints

---

In order to associate the address of the INCCOUNT() function with the vector number 69, insert the following command in the application's PRM file (Listing A.16).

#### Listing A.16 Using the VECTOR command

---

```
VECTOR 69 INCCOUNT
```

---

### Using the “interrupt Keyword”

When you are using the keyword “interrupt”, you may directly associate your interrupt function with a vector number in the ANSI C-source file. For that purpose, just specify the vector number next to the keyword interrupt.

In order to associate the address of the INCCOUNT function with the vector number 69, define the function as in Listing A.17.

#### Listing A.17 Definition of the INCCOUNT() interrupt function

---

```
interrupt 69 void INCCOUNT(void) {  
int card1;  
tcount++;  
}
```

---

## Placing an Interrupt function in a special section

For all targets supporting paging, allocate the interrupt function in an area that is accessible all the time. You can do this by placing the interrupt function in a specific segment.

### Defining a Function in a Specific Segment

In order to define a function in a specific segment, use the CODE\_SEG pragma (Listing A.18).

#### Listing A.18 Defining a Function in a Specific Segment

---

```
/* This function is defined in segment `int_Function' */  
#pragma CODE_SEG Int_Function  
#pragma TRAP_PROC  
void INCCOUNT(void) {  
    tcount++;  
}
```



```
#pragma CODE_SEG DEFAULT /* Back to default code segment.*/
```

---

## Allocating a Segment in Specific Memory

In the PRM file, you can define where you want to allocate each segment you have defined in your source code. In order to place a segment in a specific memory area, just add the segment name in the PLACEMENT block of your PRM file. Be careful, as the linker is case-sensitive. Pay special attention to the upper and lower cases in your segment name (Listing A.19).

### Listing A.19 Allocating a Segment in Specific Memory

---

```
LINK test.abs

NAMES test.o ... END

SECTIONS
    INTERRUPT_ROM = READ_ONLY    0x4000 TO 0x5FFF;
    MY_RAM        = READ_WRITE   ....

PLACEMENT
    Int_Function      INTO INTERRUPT_ROM;
    DEFAULT_RAM      INTO MY_RAM;
    ....
END
```

---

## Protecting Parameters in the OVERLAP Area

The compiler for some targets may be directed instead using memory on the stack to place the parameters or local variables into a global memory area called OVERLAP. For targets which do not have stack access for parameters or local variables, there is no other way to handle parameters and local variables.

The example below shows how to use the pragma NO\_OVERLAP and NO\_ENTRY to protect the Tim\_PresetTimer() function.. This function is called from various places, especially from an ISR. The goal is to protect the function so it is not interrupted by an interrupt. The problem is that the parameters passed to Tim\_PresetTimer() are placed in the overlap area and therefore we have to protect them else they will be overwritten. The example below shows the solution.

## Porting Tips and FAQs

### Protecting Parameters in the OVERLAP Area

---

```
#include <hidef.h>
extern char timer[];
#pragma NO_OVERLAP
#pragma NO_ENTRY
void Tim_PresetTimer(unsigned char tIndex, unsigned char PresetValue)
{
    DisableInterrupts;
    asm {
        STX    tIndex
        STA    PresetValue
    }
    timer[tIndex] = PresetValue;
    EnableInterrupts;
}
```

---

As an example the following code is given, but it looks similar for other targets:

---

```
1:  #include <hidef.h>
2:
3:  extern char timer[];
4:
5:  #pragma NO_OVERLAP
6:  #pragma NO_ENTRY
7:  void Tim_PresetTimer(unsigned char tIndex, unsigned char
PresetValue)
8:  {
9:      DisableInterrupts;
Function: Tim_PresetTimer
Options : -Ix:\chc05\lib\hc05c\include -Lasm=%n.lst
0000 9b          SEI
10:      asm {
11:          STX    tIndex
0001 cf0000     STX    _Tim_PresetTimerp1
12:          STA    PresetValue
0004 c70000     STA    _Tim_PresetTimerp0
13:      }
14:      timer[tIndex] = PresetValue;
0007 ce0000     LDX    _Tim_PresetTimerp1
000a c60000     LDA    _Tim_PresetTimerp0
000d d70000     STA    timer,X
15:      EnableInterrupts;
0010 9a          CLI
16:
17: }
```

0011 81                    RTS

---

## How to Use Variables in EEPROM

Placing variables into EEPROM is not explicitly supported in the C language. However, because EEPROM is widely available in embedded processors, a development tool for Embedded Systems must support it.

The examples are processor-specific. However, it is very easy to adapt them for any other processor.

### Linker Parameter File

You have to define your RAM or ROM areas in your linker parameter file (Listing A.20). However, you should declare the EEPROM memory as NO\_INIT to avoid initializing the memory range during normal startup.

#### Listing A.20 Linker Parameter File

---

```
LINK test.abs

NAMES test.o startup.o ansi.lib END
SECTIONS
  MY_RAM = READ_WRITE 0x800 TO 0x801;
  MY_ROM = READ_ONLY  0x810 TO 0xAFF;
  MY_STK = READ_WRITE 0xB00 TO 0xBFF;
  EEPROM = NO_INIT    0xD00 TO 0xD01;
PLACEMENT
  DEFAULT_ROM INTO MY_ROM;
  DEFAULT_RAM INTO MY_RAM;
  SSTACK      INTO MY_STK;
  EEPROM_DATA INTO EEPROM;
END

/* set reset vector to the _Startup function defined in startup code */
VECTOR ADDRESS 0xFFFFE _Startup
```

---

## The Application

The example in Listing A.21 shows an example which erases or writes an EEPROM word. The example is specific to the processor used, but it is easy to adapt if you consult the technical documentation about the EEPROM used for your derivative or CPU.

---

**NOTE** There are only a limited number of write operations guaranteed for EEPROMs so avoid writing to an EEPROM cell too frequently.

---

### Listing A.21 Erasing and Writing an EEPROM

---

```
/*
Definition of a variable in EEPROM

The variable VAR is located in EEPROM.
- It is defined in a user-defined segment EEPROM_DATA
- In the PRM file, EEPROM_DATA is placed at address 0xD00.

Be careful, the EEPROM can only be written a limited number of times.
Running this application too frequently may surpass this limit and the
EEPROM may be unusable afterwards.
*/
#include <hidef.h>
#include <stdio.h>
#include <math.h>
/* INIT register. */
typedef struct {
    union {
        struct {
            unsigned int    bit0:1;
            unsigned int    bit1:1;
            unsigned int    bit2:1;
            unsigned int    bit3:1;
            unsigned int    bit4:1;
            unsigned int    bit5:1;
            unsigned int    bit6:1;
            unsigned int    bit7:1;
        } INITEE_Bits;
        unsigned char INITEE_Byte;
    } INITEE;
} INIT;
volatile INIT INITEE @0x0012;
#define EEON INITEE.INITEE.INITEE_Bits.bit0
/* EEPROM register. */
volatile struct {
    unsigned int    EEPGM:1;
    unsigned int    EELAT:1;
```

```
    unsigned int    ERASE:1;
    unsigned int    ROW:1;
    unsigned int    BYTE:1;
    unsigned int    dummy1:1;
    unsigned int    dummy2:1;
    unsigned int    BULKP:1;
} EEPROM @0x00F3;
/* EEPROT register. */
volatile struct {
    unsigned int    BPROT0:1;
    unsigned int    BPROT1:1;
    unsigned int    BPROT2:1;
    unsigned int    BPROT3:1;
    unsigned int    BPROT4:1;
    unsigned int    dummy1:1;
    unsigned int    dummy2:1;
    unsigned int    dummy3:1;
} EEPROT @0x00F1;
#pragma DATA_SEG EEPROM_DATA
unsigned int VAR;
#pragma DATA_SEG DEFAULT
void EraseEEPROM(void) {
    /* Function used to erase one word in the EEPROM. */
    unsigned long int i;
    EEPROM.BYTE = 1;
    EEPROM.ERASE = 1;
    EEPROM.EELAT = 1;
    VAR = 0;
    EEPROM.EEPM = 1;
    for (i = 0; i<4000; i++) {
        /* Wait until EEPROM is erased. */
    }
    EEPROM.EEPM = 0;
    EEPROM.EELAT = 0;
    EEPROM.ERASE = 0;
}

void WriteEEPROM(unsigned int val) {
    /* Function used to write one word in the EEPROM. */
    unsigned long int i;
    EraseEEPROM();
    EEPROM.ERASE = 0;
    EEPROM.EELAT = 1;
    VAR = val;
    EEPROM.EEPM = 1;
    for (i = 0; i<4000; i++) {
        /* Wait until EEPROM is written. */
    }
}
```

## Porting Tips and FAQs

### General Optimization Hints

---

```
EEPROG.EEPM = 0;
EEPROG.EELAT = 0;
EEPROG.ERASE = 0;
}

void func1(void) {
    unsigned int i;
    unsigned long int ll;
    i = 0;
    do
    {
        i++;
        WriteEEPROM(i);
        for (ll = 0; ll<200000; ll++) {
        }
    }
    while (1);
}

void main(void) {
    EEPROT.BPROT4 = 0;
    EEON=1;
    WriteEEPROM(0);
    func1();
}
```

---

## General Optimization Hints

Here are some hints how to reduce the size of your application:

- Check if the full startup code is needed. e.g., if there is no initialized data, the copy-down can be ignored or removed. If the memory does not need to be initialized, the zero-out can be removed. And if both the copy-down and the zero-out are not needed, the complete startup code may be removed and the stack may be set up directly in the main routine. Set with `INIT main` in the `prm` file your main routine as application startup or entry
- Check if you need the full startup code. For example, if you do not have any initialized data, you can ignore or remove the copy-down. If you do not need any initialized memory, you can remove the zero-out. And if you do not need both, you may remove the complete startup code and directly set up your stack in your main routine. Use `INIT main` in the `prm` file as the startup or entry into your main routine of the application.

- Check the compiler options. For example, the `-OdocF`: Dynamic Option Configuration for Functions compiler option increases the compilation speed, but it decreases the code size. You can try `-OdocF="-or"`. Using the `-Li`: List of Included Files option to write a log file displays the statistics for each single option.
- Check if you can use both IEEE32 for float and double. See the `-T`: Flexible Type Management option for how to configure this. Do not forget to link the corresponding ANSI-C library.
- Use smaller data types whenever possible (e.g., 16 bits instead of 32 bits).
- Have a look into the map file to check runtime routines, which usually have a `'_'` prefix. Check for 32-bit integral routines (e.g., `_LADD`). Check if you need the long arithmetic.
- Enumerations: if you are using enums, by default they have the size of `'int'`. They can be set to an unsigned 8-bit (see option `-T`, or use `-TE1uE`).
- Check if you are using switch tables (have a look into the map file as well). There are options to configure this (see `-CswMinSLB`: Minimum Number of Labels for Search Switch Tables for an example).
- Finally, the linker has an option to overlap ROM areas (see the `-COCC` option in the linker).

## Executing an Application from RAM

For performance reasons, it may be interesting to copy an application from ROM to RAM and to execute it from RAM. This can be achieved following the procedure below.

1. Link your application with code located in RAM.
2. Generate an S-Record File.
3. Modify the startup code to copy the application code.
4. Link the application with the S-Record File previously generated.

Each step is described in the following sections. The `fibonacci.abs` application is used for an example.

Link your application with code located in RAM.

We recommend that you generate a ROM library for your application. This allows you to easily debug your final application (including the copying of the code).

### ROM Library Startup File

A ROM Library requires a very simple startup file, containing only the definition from the startup structure. Usually a ROM library startup file looks as follows:

## Porting Tips and FAQs

### Executing an Application from RAM

---

```
#include "startup.h"

/* read-only: _startupData is allocated in ROM and ROM
Library PRM File */
struct _tagStartup _startupData;
```

You must generate a PRM file to set where the code is placed in RAM. As the compiler generates absolute code, the linker should know the final location of the code in order to generate correct code for the function call.

In addition, specify the name of the application entry points in the ENTRIES block of the PRM file. The application's main function, as well as the function associated with an Interrupt vector, must be specified there.

Suppose you want to copy and execute your code at address 0x7000. Your PRM file will look as in Listing A.22.

#### Listing A.22 Linker Parameter File

---

```
LINK fiboram.abs AS ROM_LIB
NAMES myFibo.o start.o
END

SECTIONS
  MY_RAM = READ_WRITE 0x4000 TO 0x43FF;
  MY_ROM = READ_ONLY 0x7000 TO 0xBFFF; /* Dest. Address in RAM area */
PLACEMENT
  DEFAULT_ROM, ROM_VAR, STRINGS INTO MY_ROM;
  DEFAULT_RAM INTO MY_RAM;
END
ENTRIES
  myMain
END
```

---

**NOTE** You cannot use a main function in a ROM library. Please use another name for the application's entry point. In the example above, we have used "myMain".

---

## Generate an S-Record File

An S-Record File must be generated for the application. In this purpose, you can use the Burner utility.

The file is generated when you click the '1st byte(msb)' button in the burner dialog.



---

**NOTE** Initialize the field 'From' with 0 and the field 'Length' with a value bigger than the last byte used for the code. If byte 0xFFFF is used, then Length must be at least 10000.

---

## Modify the Startup Code

The startup code of the final application must be modified. It should contain code that copies the code from RAM to ROM. The application's entry point is located in the ROM library, so be sure to call it explicitly.

## Application PRM File

The S-Record File (generated previously) must be linked to the application with an offset. Suppose the application code must be placed at address 0x800 in ROM and should be copied to address 0x7000 in RAM. The application's PRM file looks as in Listing A.23.

### Listing A.23 Linker Parameter File

---

```
LINK fiboram.abs

NAMES mystart.o fiboram.abs ansis.lib END
SECTIONS

    MY_RAM = READ_WRITE 0x5000 TO 0x53FF;
    MY_ROM = READ_ONLY  0x0600 TO 0x07FF;
PLACEMENT
    DEFAULT_ROM, ROM_VAR, STRINGS INTO MY_ROM;
    DEFAULT_RAM INTO MY_RAM;
END
STACKSIZE 0x100
VECTOR 0 _Startup /* set reset vector on startup function */
HEXFILE fiboram.s1 OFFSET 0xFFFF9800 /* 0x800 - 0x7000 */
```

---

---

**NOTE** The offset specified in the HEXFILE command is added to each record in the S-Record File. The code at address 0x700 will be encoded at address 0x800.

---

If CodeWarrior is used, then the CodeWarrior IDE will pass all the names in the NAMES...END directive directly to the linker. Therefore, the NAMES...END directive should be empty.

## Copying Code from ROM to RAM

You must implement a function that copies the code from ROM to RAM.

Suppose the application code must be placed at address 0x800 in ROM and should be copied to address 0x7000 in RAM. You can implement a copy function that does this as in Listing A.24.

### Listing A.24 Definition of the CopyCode() Function

---

```
/* Start address of the application code in ROM. */
#define CODE_SRC 0x800

/* Destination address of the application code in RAM. */
#define CODE_DEST 0x7000

#define CODE_SIZE 0x90 /* Size of the code which must be copied.*/

void CopyCode(void) {
    unsigned char *ptrSrc, *ptrDest;

    ptrSrc = (unsigned char *)CODE_SRC;
    ptrDest = (unsigned char *)CODE_DEST;
    memcpy (ptrDest, ptrSrc, CODE_SIZE);
}
```

---

## Invoking the Application's Entry Point in the Startup Function

The startup code should call the application's entry point, which is located in the ROM library. You must explicitly call this function by its name. The best place is just before calling the application's main routine (Listing A.25).

### Listing A.25 Invoking the Application's Entry Point

---

```
void _Startup(void) {
    ... set up stack pointer ...
    ... zero out ...
    ... copy down ...
    CopyCode();
    ... call main ...
}
```

---

## Defining a dummy main function

The linker cannot link an application if there is no main function available. As in our case, the ROM library contains the main function. Define a dummy main function in the startup module (Listing A.26).

### Listing A.26 Definition of a dummy main Function

---

```
#pragma NO_ENTRY
#pragma NO_EXIT
void main(void) {
    asm NOP;
}
```

---

## Frequently Asked Questions (FAQs), Troubleshooting

This section provides some tips on how to solve the most commonly encountered problems.

### Making Applications

If the compiler or linker crashes, isolate the construct causing the crash and send a bug report to Freescale support. Other common problems are:

### The compiler reports an error, but WinEdit does not display it.

This means that WinEdit did not find the EDOUT file, i.e., the compiler wrote it to a place not expected by WinEdit. This can have several causes. Check that the DEFAULTDIR: Default Current Directory environment variable is not set and that the project directory is set correctly. Also in WinEdit 2.1, make sure that the OUTPUT entry in the file WINEDIT.INI is empty.

### Some programs cannot find a file.

Make sure the environment is set up correctly. Also check WinEdit's project directory. Read the Input Files section of the Files chapter.

### The compiler seems to generate incorrect code.

First, determine if the code is incorrect or not. Sometimes the operator-precedence rules of ANSI-C do not quite give the results one would expect. Sometimes faulty code can appear to be correct. Consider the example in Listing A.27:

#### Listing A.27 Possibly faulty code?

---

```
if (x & y != 0) ...
```

evaluates as:

```
if (x & (y != 0)) ...
```

but not as:

```
if ((x & y) != 0) ...
```

---

Another source of unexpected behavior can be found among the integral promotion rules of C. Characters are usually (sign-)extended to integers. This can sometimes have quite unexpected effects, e.g., the if-condition in Listing A.28 is FALSE:

#### Listing A.28 if condition is always FALSE

---

```
unsigned char a, b;  
b = -8;  
a = ~b;  
if (a == ~b) ...
```

---

because extending a results in 0x0007, while extending b gives 0x00F8 and the '~' results in 0xFF07. If the code contains a bug, isolate the construct causing it and send a bug report to Freescale support.

### The code seems to be correct, but the application does not work.

Check whether the hardware is not set up correctly (e.g., using chip selects). Some memory expansions are accessible only with a special access mode (e.g., only word accesses). If memory is accessible only in a certain way, use inline assembly or use the 'volatile' keyword.

## **The linker cannot handle an object file.**

Make sure all object files have been compiled with the latest version of the compiler and with the same flags concerning memory models and floating point formats. If not, recompile them.

## **The make utility does not make the entire application.**

Most probably you did not specify that the target is to be made on the command line. In this case, the make utility assumes the target of the first rule is the top target. Either put the rule for your application as the first in the make file, or specify the target on the command line.

## **The make utility unnecessarily re-compiler a file.**

This problem can appear if you have short source files in your application. It is caused by the fact that MS-DOS only saves the time of last modification of a file with an accuracy of  $\pm 2$  seconds. If the compiler compiles two files in that time, both will have the same time stamp. The make utility makes the safe assumption that if one file depends on another file with the same time stamp, the first file has to be recompiled. There is no way to solve this problem.

## **The help file cannot be opened by double clicking on it in the file manager or in the explorer.**

The compiler help file is a true Win32 help file. It is not compatible with the windows 3.1 version of WinHelp. The program "winhelp.exe" delivered with Windows 3.1, Windows 95 and Windows NT can only open Windows 3.1 help files. To open the compiler help file, use winhlp32.exe.

The winhlp32.exe program resides either in the windows directory (usually C:\windows, C:\win95 or C:\winnt) or in its system (Win32s) or system32 (Windows 95, 98, Me, NT, 2000, XP, or 2003) subdirectory. The Win32s distribution also contains winhlp32.exe.

To change the association with Windows 95 or Windows NT either (1) use the explorer menu "View->Options" and then the "File Types" tab or (2) select any help file and press the *Shift* key. Hold it while opening the context menu by clicking on the right mouse button. Select "Open with ..." from the menu. Enable the "Always using this program" check box and select the winhlp32.exe file with the "other" button.

## Porting Tips and FAQs

### Frequently Asked Questions (FAQs), Troubleshooting

---

To change the association with the file manager under Windows 3.1 use the “File->Associate...” menu entry.

## How can constant objects be allocated in ROM?

Use `#pragma INTO_ROM`: Put Next Variable Definition into ROM and the `-Cc`: Allocate Constant Objects into ROM compiler option.

## The compiler cannot find my source file. What is wrong?

Check if in the `default.env` file the path to the source file is set in the `GENPATH: #include “File”` Path environment variable. In addition, you can use the `-I`: Include File Path compiler option to specify the include file path. With CodeWarrior, check the access path in the preference panel.

## How can I switch off smart linking?

By adding a '+' after the object in the NAMES list of the prm file.

With CodeWarrior and the ELF/DWARF object-file format (see `-F` (`-Fh`, `-F1`, `-F1o`, `-F2`, `-F2o`, `-F6`, or `-F7`): Object-File Format) compiler option, you can link all in the object within an `ENTRIES . . . END` directive in the linker prm file:

```
ENTRIES fibo.o:* END
```

This is NOT supported in the HIWARE object-file format.

## How to avoid the ‘no access to memory’ warning?

In the simulator or debugger, change the memory configuration mode (menu Simulator > Configure) to ‘auto on access’.

## How can the same memory configuration be loaded every time the simulator or debugger is started?

Save that memory configuration under `default.mem`. For example, select Simulator->Configure-> Save and enter ‘default.mem’.

## **How can a loaded program in the simulator or debugger be started automatically and stop at a specified breakpoint?**

Define the postload.cmd file. For example:

```
bs &main t  
g
```

## **How can an overview of all the compiler options be produced?**

Type in -H: Short Help on the command line of the compiler.

## **How can a custom startup function be called after reset?**

In the prm file, use:

```
INIT myStartup
```

## **How can a custom name for the main() function be used?**

In the prm file, use:

```
MAIN myMain
```

## **How can the reset vector be set to the beginning of the startup code?**

Use this line in the prm file:

```
/* set reset vector on _Startup */  
VECTOR ADDRESS 0xFFFFE _Startup
```

### **How can the compiler be configured for the editor?**

Open the compiler, select *File > Configuration* from the menubar, and choose Editor Settings.

### **Where are configuration settings saved?**

In the `project.ini` file. With CodeWarrior, the compiler settings are stored in the `*.mcp` file.

### **What should be done when “error while adding default.env options” appears after starting the compiler?**

Choose the options set by the compiler to those set in the `default.env` file and then save them in the `project.ini` file by clicking the save button in the compiler.

### **After starting up the ICD Debugger, an "Illegal breakpoint detected" error appears. What could be wrong?**

The cable might be too long. The maximum length for unshielded cables is about 20 cm and it also depends on the electrical noise in the environment.

### **Why can no initialized data be written into the ROM area?**

The `const` qualifier must be used, and the source must be compiled with the `-Cc: Allocate Constant Objects into ROM` option.

### **Problems in the communication or losing communication.**

The cable might be too long. The maximal length for unshielded cables is about 20 cm and it also depends on the electrical noise in the environment.



## **What should be done if an assertion happens (internal error)?**

Extract the source where the assertion appears and send it as a zipped file with all the headers, options and versions of all tools.

## **How to get help on an error message?**

Either press F1 after clicking on the message to start up the help file, or else copy the message number, open the pdf manual, and make a search on the copied message number.

## **How to get help on an option?**

Open the compiler and type -H: Short Help into the command line. A list of all options appears with a short description of them. Or, otherwise, look into the manual for detailed information. A third way is to press F1 in the options setting dialog while a option is marked.

## **I cannot connect to my target board using an ICD Target Interface.**

Communication may fail for the following reasons:

- Is the parallel port working correctly? Try to print a document using the parallel port. This allows you to ensure that the parallel port is available and connected.
- Is the BDM connector designed according to the specification from P&E?
- If you are running a Windows NT or Win98 operating system, you need to install an additional driver in order to be able to communicate with the software. See section NT Installation Notice in the debugger ICD Target Interface Manual.
- The original ICD Cable from P&E should not be extended. Extending this cable can often generate communication problems. The cable should not be longer than the original 25 cm.
- Maybe the PC is too fast for the ICD cable. You can slow down the communication between the PC and the Target using the environment variable BMDELAY (e.g., `BMDELAY=50`).

## Bug Reports

If you cannot solve your problem, you may need to contact our Technical Support Department. Isolate the problem – if it is a compiler problem, write a short program reproducing the problem. Then send us a bug report.

Send or fax your bug report to your local distributor, and it will be forwarded to the Technical Support Department.

The report type gives us a clue how urgent a bug report is. The classification is:

### Information

This section describes things you would like to see improved in a future major release.

### Bug

An error for which you have a workaround or would be satisfied for the time being if we could supply a workaround. (If you already have a workaround, we'd like to know about it, too!) Of course, bugs will be fixed in the next release.

### Critical Bug

A grave error that makes it impossible for you to continue with your work.

## EBNF Notation

This chapter gives a short overview of the Extended Backus–Naur Form (EBNF) notation, which is frequently used in this document to describe file formats and syntax rules. A short introduction to EBNF is presented.

### Listing A.29 EBNF Syntax

---

```
ProcDecl  = PROCEDURE ( ArgList ).
ArgList   = Expression { , Expression }.
Expression = Term (*|/) Term.
Term      = Factor AddOp Factor.
AddOp     = + | -.
Factor    = ( [-] Number ) | ( Expression ) .
```

---

The EBNF language is a formalism that can be used to express the syntax of context-free languages. The EBNF grammar consists of a rule set called – *productions* of the form:

`LeftHandSide = RightHandSide.`

The left-hand side is a non-terminal symbol. The right-hand side describes how it is composed.

EBNF consists of the symbols discussed in the sections that follow.

- Terminal Symbols
- Non-Terminal Symbols
- Vertical Bar
- Brackets
- Parentheses
- Production End
- EBNF Syntax
- Extensions

## Terminal Symbols

Terminal symbols (terminals for short) are the basic symbols which form the language described. In above example, the word `PROCEDURE` is a terminal.

## Non-Terminal Symbols

Non-terminal symbols (non-terminals) are syntactic variables and have to be defined in a production, i.e., they have to appear on the left hand side of a production somewhere. In the example above, there are many non-terminals, e.g., `ArgList` or `AddOp`.

## Vertical Bar

The vertical bar "`|`" denotes an alternative, i.e., either the left or the right side of the bar can appear in the language described, but one of them must appear. e.g., the 3<sup>rd</sup> production above means "an expression is a term followed by either a "`*`" or a "`/`" followed by another term."

## Brackets

Parts of an EBNF production enclosed by "`[`" and "`]`" are optional. They may appear exactly once in the language, or they may be skipped. The minus sign in the last production above is optional, both `-7` and `7` are allowed.

The repetition is another useful construct. Any part of a production enclosed by "`{`" and "`}`" may appear any number of times in the language described (including zero, i.e., it

## Porting Tips and FAQs

### EBNF Notation

---

may also be skipped). `ArgList` above is an example: an argument list is a single expression or a list of any number of expressions separated by commas. (Note that the syntax in the example does not allow empty argument lists...)

## Parentheses

For better readability, normal parentheses may be used for grouping EBNF expressions, as is done in the last production of the example. Note the difference between the first and the second left bracket. The first one is part of the EBNF notation. The second one is a terminal symbol (it is quoted) and may appear in the language.

## Production End

A production is always terminated by a period.

## EBNF Syntax

The definition of EBNF in the EBNF language is:

### Listing A.30

---

```
Production = NonTerminal = Expression ..
Expression = Term { | Term}.
Term       = Factor {Factor}.
Factor     = NonTerminal
           | Terminal
           | "(" Expression ")"
           | "[" Expression "]"
           | "{" Expression }".
Terminal   = Identifier | "\"" <any char> "\".
NonTerminal = Identifier.
```

---

The identifier for a non-terminal can be any name you like. Terminal symbols are either identifiers appearing in the language described or any character sequence that is quoted.

## Extensions

In addition to this standard definition of EBNF, the following notational conventions are used.

The counting repetition: Anything enclosed by "`{`" and "`}`" and followed by a superscripted expression  $x$  must appear exactly  $x$  times.  $x$  may also be a non-terminal. In the following example, exactly four stars are allowed:

```
Stars = { "*" }4.
```

The size in bytes: Any identifier immediately followed by a number *n* in square brackets (" [ " and " ] ") may be assumed to be a binary number with the most significant byte stored first, having exactly *n* bytes. See the example in Listing A.31.

**Listing A.31 Example of a 4-byte identifier - FilePos**

---

```
Struct = RefNo FilePos[4].
```

---

In some examples, text is enclosed by "<" and ">". This text is a meta-literal, i.e., whatever the text says may be inserted in place of the text (confer <any char> in Listing A.31, where any character can be inserted).

## Abbreviations, Lexical Conventions

Table A.1 has some programming terms used in this manual.

**Table A.1 Common terminology**

Topic	Description
ANSI	American National Standards Institute
Compilation Unit	Source file to be compiled, includes all included header files
Floating Type	Numerical type with a fractional part, e.g., float, double, long double
HLL	High-level Inline Assembly
Integral Type	Numerical type without a fractional part, e.g., char, short, int, long, long long

## Number Formats

Valid constant floating number suffixes are 'F' and 'L' for float and 'l' or 'L' for long double. Note that floating constants without suffixes are double constants in ANSI. For exponential numbers 'e' or 'E' has to be used. '-' and '+' can be used for signed representation of the floating number or the exponent.

The following suffixes are supported (Table A.2):

## Porting Tips and FAQs

### Precedence and Associativity of Operators for ANSI-C

---

**Table A.2 Supported number suffixes**

Constant	Suffix	Type
floating	F	float
floating	L	long double
integral	U	unsigned int
integral	uL	unsigned long

Suffixes are not case-sensitive, e.g., ‘u1’, ‘U1’, ‘uL’ and ‘UL’ all denote an unsigned long type. Listing A.32 has examples of these numerical formats.

**Listing A.32 Examples of supported number suffixes**

---

```
+3.15f /* float */
-0.125f /* float */
3.125f /* float */
0.787F /* float */
7.125 /* double */
3.E7 /* double */
8.E+7 /* double */
9.E-7 /* double */
3.21 /* long double */
3.2e12L /* long double */
```

---

## Precedence and Associativity of Operators for ANSI-C

Table A.3 gives an overview of the precedence and associativity of operators.

**Table A.3 ANSI-C Precedence and Associativity of Operators**

Operators	Associativity
() [] -> .	left to right
! ~ ++ -- + - * & (type) sizeof	right to left
* / %	left to right

---

**Table A.3 ANSI-C Precedence and Associativity of Operators (*continued*)**

<b>Operators</b>	<b>Associativity</b>
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
	left to right
&&	left to right
	left to right
? :	right to left
= += -= *= /= %= &= ^=  = <<= >>=	right to left
,	left to right

---

**NOTE** Unary +, - and \* have higher precedence than the binary forms.

---

The precedence and associativity is determined by the ANSI-C syntax (ANSI/ISO 9899-1990, p. 38 and Kernighan/ Ritchie, “*The C Programming Language*”, Second Edition, Appendix Table 2-1).

**Listing A.33 Examples of operator precedence and associativity**

---

```
if (a == b&& c) and
if ((a == b)&& c) are equivalent.
However,
if (a == b | c)
is the same as
if ((a == b) | c)
a = b + c * d;
```

---

In Listing A.33, operator-precedence causes the product of (c\*d) to be added to b, and that sum is then assigned to a.

---

## Porting Tips and FAQs

### List of all Escape Sequences

---

In Listing A.34, the associativity rules first evaluates `c+=1`, then assigns `b` to the value of `b` plus `(c+=1)`, and then assigns the result to `a`.

#### Listing A.34 3 assignments in 1 statement

---

```
a = b += c += 1;
```

---

## List of all Escape Sequences

Table A.4 gives an overview over escape sequences which could be used inside strings (e.g., for `printf`):

**Table A.4** Escape Sequences

Description	Escape Sequence
Line Feed	<code>\n</code>
Tabulator sign	<code>\t</code>
Vertical Tabulator	<code>\v</code>
Backspace	<code>\b</code>
Carriage Return	<code>\r</code>
Line feed	<code>\f</code>
Bell	<code>\a</code>
Backslash	<code>\\</code>
Question Mark	<code>\?</code>
Quotation Mark	<code>\^</code>
Double Quotation Mark	<code>\"</code>
Octal Number	<code>\ooo</code>
Hexadecimal Number	<code>\xhh</code>



# Global Configuration-File Entries

---

This appendix documents the entries that can appear in the global configuration file. This file is named `mcutools.ini`.

`mcutools.ini` can contain these sections:

- [Options] Section
- [XXX\_Compiler] Section
- [Editor] Section
- Example

## [Options] Section

This section documents the entries that can appear in the [Options] section of the file `mcutools.ini`.

---

### DefaultDir

#### Arguments

Default directory to be used.

#### Description

Specifies the current directory for all tools on a global level (see also environment variable `DEFAULTDIR`: Default Current Directory).

#### Example

```
DefaultDir=C:\install\project
```

## Global Configuration-File Entries

[XXX\_Compiler] Section

---

# [XXX\_Compiler] Section

This section documents the entries that can appear in an [XXX\_Compiler] section of the `mcutools.ini` file.

---

**NOTE** XXX is a placeholder for the name of the actual backend. For example, for the HC08 compiler, the name of this section would be [HC08\_Compiler].

---

---

## SaveOnExit

### Arguments

1/0

### Description

Set to 1 if the configuration should be stored when the compiler is closed. Set to 0 if it should not be stored. The compiler does not ask to store a configuration in either case.

---

## SaveAppearance

### Arguments

1/0

### Description

Set to 1 if the visible topics should be stored when writing a project file. Set to 0 if not. The command line, its history, the windows position, and other topics belong to this entry.

---

## SaveEditor

### Arguments

1/0

**Description**

Set to 1 if the visible topics should be stored when writing a project file. Set to 0 if not. The editor setting contains all information of the *Editor Configuration* dialog box.

---

**SaveOptions**

**Arguments**

1/0

**Description**

Set to 1 if the options should be saved when writing a project file. Set to 0 if the options should not be saved. The options also contain the message settings.

---

**RecentProject0, RecentProject1, ...**

**Arguments**

Names of the last and prior project files

**Description**

This list is updated when a project is loaded or saved. Its current content is shown in the file menu.

**Example**

```
SaveOnExit=1
SaveAppearance=1
SaveEditor=1
SaveOptions=1
RecentProject0=C:\myprj\project.ini
RecentProject1=C:\otherprj\project.ini
```

## Global Configuration-File Entries

[XXX\_Compiler] Section

---

### TipFilePos

#### Arguments

Any integer, e.g., 236

#### Description

Actual position in tip of the day file. Used that different tips are shown at different calls.

#### Saved

Always saved when saving a configuration file.

---

### ShowTipOfDay

#### Arguments

0 / 1

#### Description

Should the *Tip of the Day* dialog box be shown at startup.

- 1: It should be shown
- 0: Only when opened in the help menu

#### Saved

Always saved when saving a configuration file.

---

### TipTimeStamp

#### Arguments

date and time

#### Description

Date and time when the tips were last used.

---

### Saved

Always saved when saving a configuration file.

## [Editor] Section

This section documents the entries that can appear in the [Editor] section of the `mcutools.ini` file.

---

### Editor\_Name

#### Arguments

The name of the global editor

#### Description

Specifies the name which is displayed for the global editor. This entry has only a descriptive effect. Its content is not used to start the editor.

#### Saved

Only with *Editor Configuration* set in the *File > Configuration Save Configuration* dialog box.

---

### Editor\_Exe

#### Arguments

The name of the executable file of the global editor

#### Description

Specifies the filename that is called (for showing a text file) when the global editor setting is active. In the *Editor Configuration* dialog box, the global editor selection is active only when this entry is present and not empty.

#### Saved

Only with *Editor Configuration* set in the *File > Configuration Save Configuration* dialog box.

---

## Global Configuration-File Entries

Example

---

### Editor\_Opts

#### Arguments

The options to use the global editor

#### Description

Specifies options used for the global editor. If this entry is not present or empty, “%f” is used. The command line to launch the editor is built by taking the Editor\_Exe content, then appending a space followed by this entry.

#### Saved

Only with *Editor Configuration* set in the *File > Configuration Save Configuration* dialog box.

#### Example

```
[Editor]
editor_name=notepad
editor_exe=C:\windows\notepad.exe
editor_opts=%f
```

## Example

Listing B.1 shows a typical `mcutools.ini` file.

### Listing B.1 Typical `mcutools.ini` file layout

---

```
[Installation]
Path=c:\Freescale
Group=ANSI-C Compiler

[Editor]
editor_name=notepad
editor_exe=C:\windows\notepad.exe
editor_opts=%f

[Options]
DefaultDir=c:\myprj

[XXXX_Compiler]
```

---

## Global Configuration-File Entries

*Example*

---

```
SaveOnExit=1
SaveAppearance=1
SaveEditor=1
SaveOptions=1
RecentProject0=c:\myprj\project.ini
RecentProject1=c:\otherprj\project.ini
TipFilePos=0
ShowTipOfDay=1
TipTimeStamp=Jan 21 2006 17:25:16
```

---

## Global Configuration-File Entries

*Example*

---



# Local Configuration-File Entries

---

This appendix documents the entries that can appear in the local configuration file. Usually, you name this file `project.ini`, where `project` is a placeholder for the name of your project.

A `project.ini` file can contain these sections:

- [Editor] Section
- [XXX\_Compiler] Section
- Example

## [Editor] Section

---

### Editor\_Name

#### Arguments

The name of the local editor

#### Description

Specifies the name that is displayed for the local editor. This entry contains only a descriptive effect. Its content is not used to start the editor.

#### Saved

Only with *Editor Configuration* set in the *File > Configuration Save Configuration* dialog box. This entry has the same format as the global *Editor Configuration* in the `mcutools.ini` file.

## Local Configuration-File Entries

[Editor] Section

---

### Editor\_Exe

#### Arguments

The name of the executable file of the local editor

#### Description

Specifies the filename that is used for a text file when the local editor setting is active. In the *Editor Configuration* dialog box, the local editor selection is only active when this entry is present and not empty.

#### Saved

Only with *Editor Configuration* set in the *File > Configuration Save Configuration* dialog box. This entry has the same format as for the global *Editor Configuration* in the `mcutools.ini` file.

---

### Editor\_Opts

#### Arguments

Local editor options

#### Description

Specifies options that should be used for the local editor. If this entry is not present or empty, “%f” is used. The command line to launch the editor is built by taking the `Editor_Exe` content, then appending a space followed by this entry.

#### Saved

Only with *Editor Configuration* set in the *File > Configuration Save Configuration* dialog box. This entry has the same format as the global Editor Configuration in the `mcutools.ini` file.

---

### Example [Editor] Section

```
[Editor]
editor_name=notepad
editor_exe=C:\windows\notepad.exe
```

---

editor\_opts=%f

## [XXX\_Compiler] Section

This section documents the entries that can appear in an [XXX\_Compiler] section of a *project.ini* file.

---

**NOTE** *XXX* is a placeholder for the name of the actual backend. For example, for the HC08 compiler, the name of this section would be [HC08\_Compiler].

---

---

## RecentCommandLineX

---

**NOTE** *X* is a placeholder for an integer.

---

### Arguments

String with a command line history entry, e.g., “*fibonacci.c*”

### Description

This list of entries contains the content of the command line history.

### Saved

Only with *Appearance* set in the *File > Configuration Save Configuration* dialog box.

---

## CurrentCommandLine

### Arguments

String with the command line, e.g., “*fibonacci.c -w1*”

### Description

The currently visible command line content.

## Local Configuration-File Entries

[XXX\_Compiler] Section

---

### Saved

Only with *Appearance* set in the *File > Configuration Save Configuration* dialog box.

---

## StatusbarEnabled

### Arguments

1/0

### Special

This entry is only considered at startup. Later load operations do not use it afterwards.

### Description

Is status bar currently enabled.

- 1: The status bar is visible
- 0: The status bar is hidden

### Saved

Only with *Appearance* set in the *File > Configuration Save Configuration* dialog box.

---

## ToolbarEnabled

### Arguments

1/0

### Special

This entry is only considered at startup. Later load operations do not use it afterwards.

### Description

Is the toolbar currently enabled.

- 1: The toolbar is visible
-

- 0: The toolbar is hidden

### **Saved**

Only with *Appearance* set in the *File > Configuration Save Configuration* dialog box.

---

## **WindowPos**

### **Arguments**

10 integers, e.g., “0, 1, -1, -1, -1, -1, 390, 107, 1103, 643”

### **Special**

This entry is only considered at startup. Later load operations do not use it afterwards.

Changes of this entry do not show the “\*” in the title.

### **Description**

This number contains the position and the state of the window (maximized) and other flags.

### **Saved**

Only with *Appearance* set in the *File > Configuration Save Configuration* dialog box.

---

## **WindowFont**

### **Arguments**

---

size: == 0 -> generic size, < 0 -> font character height,  
> 0 -> font cell height  
weight: 400 = normal, 700 = bold (valid values are 0 - 1000)  
italic: 0 == no, 1 == yes  
font name: max 32 characters

---

## Local Configuration-File Entries

[XXX\_Compiler] Section

---

### Description

Font attributes.

### Saved

Only with *Appearance* set in the *File > Configuration Save Configuration* dialog box.

### Example

```
WindowFont=-16,500,0,Courier
```

---

## Options

### Arguments

-W2

### Description

The currently active option string. This entry is quite long as the messages are also stored here.

### Saved

Only with *Options* set in the *File > Configuration Save Configuration* dialog box.

---

## EditorType

### Arguments

0/1/2/3

### Description

This entry specifies which Editor Configuration is active.

- 0: Global Editor Configuration (in the file `mcutools.ini`)
  - 1: Local Editor Configuration (the one in this file)
  - 2: Command line Editor Configuration, entry `EditorCommandLine`
  - 3: DDE Editor Configuration, entries beginning with `EditorDDE`
-

For details see Editor Settings dialog box.

**Saved**

Only with *Editor Configuration* set in the *File > Configuration Save Configuration* dialog box.

---

**EditorCommandLine**

**Arguments**

Command line for the editor.

**Description**

Command line content to open a file. For details see Editor Settings dialog box.

**Saved**

Only with *Editor Configuration* set in the *File > Configuration Save Configuration* dialog box.

---

**EditorDDEClientName**

**Arguments**

Client command, e.g., “[open (%f) ]”

**Description**

Name of the client for DDE Editor Configuration. For details see Editor Started with DDE.

**Saved**

Only with *Editor Configuration* set in the *File > Configuration Save Configuration* dialog box.

---

## Local Configuration-File Entries

*Example*

---

### EditorDDETopicName

#### Arguments

Topic name. For example, “system”

#### Description

Name of the topic for DDE Editor Configuration. For details, see  
Editor Started with DDE

#### Saved

Only with *Editor Configuration* set in the *File > Configuration Save Configuration* dialog box.

---

### EditorDDEServiceName

#### Arguments

Service name. For example, “system”

#### Description

Name of the service for DDE Editor Configuration. For details, see  
Editor Started with DDE.

#### Saved

Only with *Editor Configuration* set in the *File > Configuration Save Configuration* dialog box.

---

## Example

Listing C.1 shows a typical configuration file layout (usually *project.ini*):

### Listing C.1 A Typical Local Configuration File Layout

---

```
[Editor]
Editor_Name=notepad
Editor_Exe=C:\windows\notepad.exe
```

---



## Local Configuration-File Entries

*Example*

---

```
Editor_Opts=%f

[XXX_Compiler]
StatusBarEnabled=1
ToolbarEnabled=1
WindowPos=0,1,-1,-1,-1,-1,390,107,1103,643
WindowFont=-16,500,0,Courier
Options=-w1
EditorType=3
RecentCommandLine0=fibo.c -w2
RecentCommandLine1=fibo.c
CurrentCommandLine=fibo.c -w2
EditorDDEClientName=[open(%f)]
EditorDDETopicName=system
EditorDDEServiceName=msdev
EditorCommandLine=C:\windows\notepad.exe %f
```

---

## Local Configuration-File Entries

*Example*

---

# Index

## Symbols

-! 143  
# 398  
## 398, 440  
#asm 413  
#define 205, 398  
#elif 398  
#else 398  
#endasm 413  
#endif 398  
#error 398, 400  
#if 398  
#ifdef 398  
#ifndef 398  
#include 207, 398  
#line 398  
#pragma 398  
    CODE\_SECTION 428  
    CODE\_SEG 357, 428, 473  
    CONST\_SECTION 155, 428  
    CONST\_SEG 360, 428  
    CREATE\_ASM\_LISTING 363  
    DATA\_SECTION 428  
    DATA\_SEG 364, 428  
    FAR 473  
    IGNORE\_DSR 367  
    INLINE 247, 367  
    INTO\_ROM 155, 368  
    LINK\_INFO 370  
    LOOP\_UNROLL 372  
    mark 373  
    MESSAGE 375  
    NEAR 473  
    NO\_ENTRY 377, 484  
    NO\_EXIT 379  
    NO\_FRAME 381  
    NO\_INLINE 383  
    NO\_LOOP\_UNROLL 384  
    NO\_PARREGS 385  
    NO\_STRING\_CONSTR 387, 439  
    ONCE 388  
    OPTION 342, 389  
    SHORT 473  
    STACK\_TEST 392  
    STRING\_SEG 392  
    TEST\_CODE 394  
    TRAP\_PROC 396, 412  
#pragma CONST\_SEG 671  
#pragma DATA\_SEG 671  
#pragma section 671  
#undef 398  
#warning 398, 400  
\$ 399  
\$( ) 111  
\${ } 111  
%(ENV) 141  
%" 141  
'% 141  
%currentTargetName 67  
%E 141  
%e 141  
%f 141  
%N 140  
%n 141  
%p 140  
%projectFileDir 67  
%projectFileName 67  
%projectFilePath 66  
%projectSelectedFiles 67  
%sourceFileDir 66  
%sourceFileName 66  
%sourceFilePath 66  
%sourceLineNumber 66  
%sourceSelection 66  
%sourceSelUpdate 66  
%symFileDir 67  
%symFileName 67  
%symFilePath 67  
%targetFileDir 67  
%targetFileName 67  
%targetFilePath 67  
\*.bbl 68  
.abs 61  
.c 131

---

.h 131  
 .ini 86  
 .lcf 674  
 .lib 65  
 .lst 496  
 .mcp 696  
 .o 132  
 /wait 81  
 @ “SegmentName” 401  
 @address 400  
 @bool 676  
 @far 676  
 @interrupt 678  
 @tiny 676  
 \_\_alignof\_\_ 399, 410  
 \_\_asm 146, 399, 481  
     asm 412  
 \_\_BIG\_ENDIAN\_\_ 340  
 \_\_BIT\_SEG 357, 360  
 \_\_BITFIELD\_TYPE\_SIZE\_REDUCTION\_\_ 153  
 \_\_BITFIELD\_LSBIT\_FIRST\_\_ 149, 346, 350, 675  
 \_\_BITFIELD\_LSBYTE\_FIRST\_\_ 149, 350, 675  
 \_\_BITFIELD\_LSWORD\_FIRST\_\_ 149, 346, 350  
 \_\_BITFIELD\_MSBIT\_FIRST\_\_ 149, 346, 350, 675  
 \_\_BITFIELD\_MSBYTE\_FIRST\_\_ 149, 346, 350, 675  
 \_\_BITFIELD\_MWORD\_FIRST\_\_ 149, 346, 350  
 \_\_BITFIELD\_NO\_TYPE\_SIZE\_REDUCTION\_\_ 153, 348  
 \_\_BITFIELD\_TYPE\_SIZE\_REDUCTION\_\_ 348  
 \_\_CHAR\_IS\_16BIT\_\_ 293, 350  
 \_\_CHAR\_IS\_32BIT\_\_ 293, 350  
 \_\_CHAR\_IS\_64BIT\_\_ 293, 350  
 \_\_CHAR\_IS\_8BIT\_\_ 293, 350  
 \_\_CHAR\_IS\_SIGNED\_\_ 293, 350  
 \_\_CHAR\_IS\_UNSIGNED\_\_ 293, 350  
 \_\_CNI\_\_ 163, 341  
 \_\_CODE\_SEG 357, 360, 364, 392  
 \_\_DATE\_\_ 339  
 \_\_DEMO\_MODE\_\_ 340  
 \_\_DIRECT\_SEG 357, 360, 364, 392  
 \_\_DOUBLE\_IS\_DSP\_\_ 294, 352  
 \_\_DOUBLE\_IS\_IEEE32\_\_ 294, 351  
 \_\_DOUBLE\_IS\_IEEE64\_\_ 294, 351  
 \_\_ELF\_OBJECT\_FILE\_FORMAT\_\_ 192, 346  
 \_\_ENUM\_IS\_16BIT\_\_ 294, 351  
 \_\_ENUM\_IS\_32BIT\_\_ 294, 351  
 \_\_ENUM\_IS\_64BIT\_\_ 294, 351  
 \_\_ENUM\_IS\_8BIT\_\_ 294, 351  
 \_\_ENUM\_IS\_SIGNED\_\_ 294, 351  
 \_\_ENUM\_IS\_UNSIGNED\_\_ 294, 351  
 \_\_far 398, 404, 464  
     Arrays 405  
     Keyword 404  
 \_\_FAR\_SEG 357, 360, 364, 392  
 \_\_FILE\_\_ 339  
 \_\_FLOAT\_IS\_DSP\_\_ 294, 351  
 \_\_FLOAT\_IS\_IEEE32\_\_ 294, 351  
 \_\_FLOAT\_IS\_IEEE64\_\_ 294, 351  
 \_\_HC08\_\_ 352  
 \_\_HCS08\_\_ 352  
 \_\_HIWARE\_\_ 340  
 \_\_HIWARE\_OBJECT\_FILE\_FORMAT\_\_ 192, 346  
 \_\_INT\_IS\_16BIT\_\_ 293, 351  
 \_\_INT\_IS\_32BIT\_\_ 293, 351  
 \_\_INT\_IS\_64BIT\_\_ 294, 351  
 \_\_INT\_IS\_8BIT\_\_ 293, 351  
 \_\_Interrupt 399  
 \_\_interrupt 412  
 \_\_LINE\_\_ 339  
 \_\_LITTLE\_ENDIAN\_\_ 340  
 \_\_LONG\_DOUBLE\_IS\_DSP\_\_ 294, 352  
 \_\_LONG\_DOUBLE\_IS\_IEEE32\_\_ 294, 352  
 \_\_LONG\_DOUBLE\_IS\_IEEE64\_\_ 294, 352  
 \_\_LONG\_IS\_16BIT\_\_ 294, 351  
 \_\_LONG\_IS\_32BIT\_\_ 294, 351  
 \_\_LONG\_IS\_64BIT\_\_ 294, 351  
 \_\_LONG\_IS\_8BIT\_\_ 294, 351  
 \_\_LONG\_LONG\_DOUBLE\_DSP\_\_ 294, 352  
 \_\_LONG\_LONG\_DOUBLE\_IS\_IEEE32\_\_ 294, 352

---

---

\_\_LONG\_LONG\_DOUBLE\_IS\_IEEE64\_\_ 294, 352  
 \_\_LONG\_LONG\_IS\_16BIT\_\_ 294, 351  
 \_\_LONG\_LONG\_IS\_32BIT\_\_ 294, 351  
 \_\_LONG\_LONG\_IS\_64BIT\_\_ 294, 351  
 \_\_LONG\_LONG\_IS\_8BIT\_\_ 294, 351  
 \_\_MODULO\_IS\_POSITIV\_\_ 346  
 \_\_MWERKS\_\_ 340  
 near 398, 409, 464  
 \_\_NEAR\_SEG 357, 360, 364, 392  
 \_\_NO\_RECURSION\_\_ 352  
 \_\_OPTIMIZE\_FOR\_SIZE\_\_ 234, 341  
 \_\_OPTIMIZE\_FOR\_TIME\_\_ 234, 341  
 \_\_OPTIMIZE\_REG\_\_ 281  
 \_\_OPTION\_ACTIVE\_\_ 341  
 \_\_PLAIN\_BITFIELD\_IS\_SIGNED\_\_ 294, 349, 350, 352  
 \_\_PLAIN\_BITFIELD\_IS\_UNSIGNED\_\_ 294, 349, 350, 352  
 \_\_PRODUCT\_HICROSS\_PLUS\_\_ 340  
 \_\_PTR\_SIZE\_1\_\_ 352  
 \_\_PTR\_SIZE\_2\_\_ 353  
 \_\_PTR\_SIZE\_3\_\_ 353  
 \_\_PTR\_SIZE\_4\_\_ 353  
 \_\_PTRDIFF\_T\_IS\_CHAR\_\_ 344, 345  
 \_\_PTRDIFF\_T\_IS\_INT\_\_ 344, 345  
 \_\_PTRDIFF\_T\_IS\_LONG\_\_ 344, 345  
 \_\_PTRDIFF\_T\_IS\_SHORT\_\_ 344, 345  
 \_\_PTRMBR\_OFFSET\_IS\_16BIT\_\_ 294  
 \_\_PTRMBR\_OFFSET\_IS\_32BIT\_\_ 294  
 \_\_PTRMBR\_OFFSET\_IS\_64BIT\_\_ 295  
 \_\_PTRMBR\_OFFSET\_IS\_8BIT\_\_ 294  
 \_\_SHORT\_IS\_16BIT\_\_ 293, 350  
 \_\_SHORT\_IS\_32BIT\_\_ 293, 351  
 \_\_SHORT\_IS\_64BIT\_\_ 293, 351  
 \_\_SHORT\_IS\_8BIT\_\_ 293, 350  
 \_\_SHORT\_SEG 360, 364, 428, 452  
 \_\_SIZE\_T\_IS\_UCHAR\_\_ 343, 344  
 \_\_SIZE\_T\_IS\_UINT\_\_ 343, 344  
 \_\_SIZE\_T\_IS\_ULONG\_\_ 343, 344  
 \_\_SIZE\_T\_IS\_USHORT\_\_ 343, 344  
 \_\_SMALL\_\_ 226, 352  
 \_\_STDC\_\_ 146, 339, 341  
 \_\_TIME\_\_ 339  
 \_\_TINY\_\_ 226, 352  
 \_\_TRIGRAPHS\_\_ 160, 341  
 \_\_va\_sizeof\_\_ 399, 411  
 \_\_VERSION\_\_ 340  
 \_\_VTAB\_DELTA\_IS\_16BIT\_\_ 294, 352  
 \_\_VTAB\_DELTA\_IS\_32BIT\_\_ 294, 352  
 \_\_VTAB\_DELTA\_IS\_64BIT\_\_ 294, 352  
 \_\_VTAB\_DELTA\_IS\_8BIT\_\_ 294, 352  
 \_\_WCHAR\_T\_IS\_UCHAR\_\_ 344  
 \_\_WCHAR\_T\_IS\_UINT\_\_ 344  
 \_\_WCHAR\_T\_IS\_ULONG\_\_ 344  
 \_\_WCHAR\_T\_IS\_USHORT\_\_ 344  
 \_asm 399, 677  
 \_IOFBF 517  
 \_IOLBF 517  
 \_IONBF 517  
 \_STACK 487  
 {Compiler} 111  
 {Project} 111  
 {System} 111

## Numerics

0b 399

## A

abort 498, 524  
 About 68, 69, 70, 71, 73, 74  
 About Box 106  
 abs 525  
 absolute assembly application 74  
 Absolute Functions 403  
 absolute variables 400  
 Absolute Variables and Linking 403  
 ABSPATH 94  
 acosf 526  
 Add Additional Files dialog box 32  
 -AddIncl 144  
 adding 61  
 ahc08.exe 59  
 Alignment 466, 469  
 alloc.c 497  
 -Ansi 146, 339, 341  
 ANSI-C 163, 164, 342  
 Reference Document 397

---

---

Standard 397  
ANSI-C 397  
Application File Name 73  
Array  
    \_\_far 405  
Arrays with unknown size 676  
asctime 527  
asin 528  
asinf 528  
asm 146, 399, 413, 481  
-Asr 147  
Assembler 481  
Assembler for HC08 preference panel 67  
assert 529  
assert.h 519  
Associativity 702  
atan 530  
atan2 531  
atan2f 531  
atanf 530  
atexit 498, 532  
atof 533  
atoi 534  
atol 535  
auto 397

## **B**

batch file 80  
-BfaB 149, 350  
-BfaGapLimitBits 151  
-BfaTSR 153  
-BfaTSROFF 348  
-BfaTSRON 348  
Big Endian 340  
bin 60  
Binary Constants 399  
binplugins 60  
BIT 360  
Bit Fields 426, 466, 675  
Branch Optimization 431, 475  
Branch Sequence 433  
Branch Tree 433  
break 397  
browse information 66

bsearch 536  
BUFSIZ 517  
Build Extras preference panel 65, 66  
Burner 69  
Burner for HC08 preference Panel 69  
Burner for HC08 preference panel 68  
burner.exe 59

## **C**

C++ comments 146, 166  
C/C++ Options panel 34  
Caller/Callee Saved Registers 484  
calloc 497, 538  
case 397  
-Cc 155, 361, 367, 430, 694, 696  
-Ccx 157, 669, 673  
ceil 539  
ceilf 539  
-Cf 160  
char 397, 414  
CHAR\_BIT 512  
CHAR\_MAX 512  
CHAR\_MIN 512  
chc08.exe 59  
-Ci 160, 341  
clearerr 540  
ClientCommand 90  
clock 541  
clock\_t 519  
CLOCKS\_PER\_SEC 519  
-Cni 163, 341  
-CnMUL 163  
CODE 138, 357, 360, 364, 392  
CODE GENERATION 140  
Code Size 418  
CODE\_SECTION 357, 428  
CODE\_SEG 357, 428  
CodeWarrior 58, 60, 91, 689, 694, 696  
CodeWarrior project window 36  
CodeWright 89  
color 305, 306, 307, 308, 309  
COM 60, 91  
Command Line Arguments 68, 69, 70, 71, 72, 73  
comments 677

---

Common Source Files 493  
 Compiler  
     Configuration 86  
     Control 101  
     Error  
         Messages 106  
     Error Feedback 107  
     Include file 131  
     Input File 106, 131  
     Menu 96  
     Menu Bar 85  
     Messages 103  
     Option 99  
     Option Settings Dialog 99  
     Standard Types Dialog Box 98  
     Status Bar 85  
     Tool Bar 84  
 Compiler for HC08 preference panel 69, 70  
 COMOPTIONS 113, 116, 135  
 const 155, 397, 434  
 CONST\_SECTION 155, 360, 428  
 CONST\_SEG 360, 428  
 Constant Function 454  
 continue 397  
 Copy Down 494  
 copy down 404  
 Copy Template 73  
 Copying Code from ROM to RAM 690  
 COPYRIGHT 117  
 cos 542  
 cosf 542  
 cosh 543  
 coshf 543  
 Cosmic 669  
 -Cp 166  
 -Cpcc 166  
 -CPU 168  
 -Cpu 168  
 -Cq 168, 171  
 CREATE\_ASM\_LISTING 363  
 -Cs08 170, 470, 471, 495  
 -CsIni0 170  
 -CswMaxLF 171  
 -CswMinLB 173  
 -CswMinLF 175  
 -CswMinSLB 177, 687  
 ctime 544  
 CTRL-S 95  
 ctype 500  
 ctype.h 521  
 -Cu 137, 179, 372, 384  
 Current Directory 110, 118  
 CurrentCommandLine 715  
 -Cv 182  
 -Cx 182

**D**

-D 183  
 DATA\_SECTION 364, 428  
 DATA\_SEG 364, 428, 452  
 decoder.exe 59  
 default 397  
 Default Directory 705  
 DEFAULT.ENV 110, 118, 119, 127  
 default.env 135  
 DEFAULTDIR 111, 118, 131  
 DefaultDir 705  
 define 183  
 defined 398  
 Device and Connection dialog box 31  
 difftime 545  
 DIG 512  
 DIRECT 357, 360, 364, 392  
 Directive  
     #define 205, 398  
     #elif 398  
     #else 398  
     #endif 398  
     #error 398  
     #if 398  
     #ifdef 398  
     #ifndef 398  
     #include 207, 398  
     #line 398  
     #pragma 398  
     #undef 398  
     Preprocessor 397

---

---

Display generated command lines in message  
     window 68, 69, 71, 72, 74  
 div 546  
 div\_t 518  
 Division 345, 414  
 do 397  
 DOS 143  
 double 397  
 download 404

**E**

EABI 349  
 EBNF 698  
 -Ec 185  
 Editor 713  
 Editor\_Exe 709, 714  
 Editor\_Name 709, 713  
 Editor\_Opts 710, 714  
 EditorCommandLine 719  
 EditorDDEClientName 719  
 EditorDDEServiceName 720  
 EditorDDETopicName 720  
 EditorType 718  
 EDOM 511  
 EDOUT 132  
 -Eencrypt 187  
 EEPROM 683  
 -Ekey 189  
 ELF/DWARF 77, 98, 403, 694  
 ELF/DWARF object-file format 77  
 else 397  
 Embedded Application Binary Interface 349  
 Endian 340  
 ENTRIES 403  
 Entry Code 471  
 enum 397  
 -Env 190  
 ENVIRONMENT 110, 119  
 Environment  
     COMPOPTIONS 116, 135  
     COPYRIGHT 117  
     DEFAULTDIR 111, 118, 131  
     ENVIRONMENT 110, 119  
     ENVIRONMENT 109  
     ERRORFILE 120  
     File 110  
     GENPATH 122, 124, 125, 131, 197  
     HICOMPOPTIONS 116  
     HIENVIRONMENT 119  
     HIPATH 122, 125  
     INCLUDETIME 123  
     LIBPATH 122, 124, 128, 131, 132, 197, 198  
     LIBRARYPATH 124, 131, 132, 197, 198  
     OBJPATH 125, 132  
     TEXTPATH 126, 199, 214, 221  
     TMP 127  
     USELIBPATH 128  
     USERNAME 129  
     Variable 115  
 Environment Variable 109, 355  
     110  
 Environment Variables 94  
 EOF 517  
 EPROM 404  
 EPSILON 512  
 ERANGE 511  
 errno 511  
 errno.h 511  
 Error  
     Handling 501  
     Listing 132  
     Messages 106  
 Error Format  
     Microsoft 312  
     Verbose 312  
 Error Listing 132  
 ERRORFILE 120  
 Escape Sequences 704  
 exit 498, 547  
 Exit Code 472  
 EXIT\_FAILURE 518  
 EXIT\_SUCCESS 518  
 exp 548  
 expf 548  
 Explorer 79, 110  
 Extended Backus-Naur Form, see EBNF  
 extern 397

---



---

## F

-F1 192, 346, 429  
-F1o 192  
-F2 192, 346, 429  
F2 84  
-F2o 192  
-F6 192  
-F7 192  
fabs 249, 549  
fabsf 249, 549  
FAR 357, 360, 364, 392, 473  
far 398, 404  
fclose 550  
-Fd 194  
feof 551  
ferror 552  
fflush 553  
fgetc 554  
fgetpos 555  
fgets 556  
-Fh 192, 346  
FILE 517  
File  
    Environment 110  
    Include 131  
    Object 132  
    Source 131  
File Manager 110  
File Names 418  
FILENAME\_MAX 517  
float 397  
float.h 511  
Floating Point 465  
floor 557  
floorf 557  
FLT\_RADIX 511  
FLT\_ROUNDS 511  
fmod 558  
fopen 559  
FOPEN\_MAX 517  
for 397  
fpos\_t 517  
fprintf 561  
fputc 562

fputs 563  
Frame Pointer 471  
Frame, see Stack Frame  
fread 564  
free 497, 565  
freopen 566  
frexp 567  
frexpf 567  
Front End 397  
fscanf 568  
fseek 569  
fsetpos 570  
ftell 571  
fwrite 572

## G

GENPATH 94, 122, 124, 125, 131, 197  
getc 573  
getchar 574  
getenv 575  
gets 576  
gmtime 577  
groups, CodeWarrior 39

## H

-H 195, 695, 697  
HALT 497, 498  
HC12 Compiler Option Settings dialog box 47  
heap.c 497  
Help 68, 69, 70, 71, 73, 74  
Hexadecimal Constants 399  
HICOMPOPTIONS 116  
HIENVIRONMENT 119  
High Address Part 488  
HIPATH 122  
HIWARE object-file format 77  
hiwave.exe 59, 66  
HOST 138, 140  
How to Generate Library 493  
HUGE\_VAL 515

## I

-I 131, 197, 694

---

I/O Registers 404

ICD 697

Icon 80

ide.exe 58

IEEE 465

if 397

IGNORE\_DSR 367

Implementation Restriction 415

Importer for HC08 preference panel 71

Include Files 131, 417

INCLUDETIME 123

INLINE 247, 367

inline 247, 451

Inline Assembler, see Assembler

INPUT 138, 140

int 397

INT\_MAX 513

INT\_MIN 513

Intel 340

Internal ID's 418

Interrupt 412, 472, 678

    keyword 412

    vector 412

interrupt 399, 464

INTO\_ROM 155, 368

IPATH 125

isalnum 578

isalpha 578

iscentrl 578

isdigit 578

isgraph 578

islower 578

isprint 578

ispunct 578

isspace 578

isupper 578

isxdigit 578

## J

jmp\_buf 515

Jump Table 433

## K

Keywords 463

## L

-La 199

Labels 417

labs 580

LANGUAGE 140

-Lasm 201

-Lasmc 203

Lazy Instruction Selection 474

lconv 513

ldexp 581

ldexpf 581

-Ldf 205, 339

ldiv 582

ldiv\_t 518

Lexical Tokens 417

-Li 207

libmaker 65

Libmaker for HC08 preference panel 73

libmaker.exe 59

LIBPATH 94, 122, 124, 128, 131, 132, 197, 198

library 65

Library File Name 74

Library Files 493, 495

LIBRARYPATH 124, 131, 132, 197, 198

-Lic 209, 211

-LicA 210, 213

Limits

    Translation 415

limits.h 512

Line Continuation 114

LINK\_INFO 370

Linker for HC08 preference panel 72

linker.exe 59

Little Endian 340

-Ll 214

-Lm 216

-LmCfg 218

-Lo 221

locale.h 513

localeconv 583

Locales 500

localtime 584

log 585

log10 586

---

log10f 586  
logf 585  
long 397  
LONG\_MAX 513  
LONG\_MIN 513  
longjmp 587  
LOOP\_UNROLL 372  
-Lp 222  
-LpCfg 223  
-LpX 225

## M

Macro 183  
    Predefined 339  
Macro Expansion 417  
maker.exe 59  
malloc 497, 588  
MANT\_DIG 512  
mark 373  
math.h 515, 624  
MAX 512  
MAX\_10\_EXP 512  
MAX\_EXP 512  
MB\_LEN\_MAX 513, 518  
mblen 498, 589  
mbstowcs 498, 590  
mbtowc 498, 591  
MCUTOOLS.INI 87, 112  
memchr 592  
memcmp 593  
memcpy 249, 594  
memmove 594  
Memory Models 463  
memset 249, 595  
MESSAGE 140, 375  
MESSAGES 139  
Messages 68, 69, 70, 71, 72, 73  
Microsoft 312  
Microsoft Developer Studio 90  
Microsoft Visual Studio 74  
MIN 512  
MIN\_10\_EXP 512  
MIN\_EXP 512  
Missing Prototype 677

mktime 596  
modf 597  
modff 597  
Modulus 345, 414  
-Ms 226, 352, 463, 469, 495  
MSB 488  
msdev 90  
MS-DOS 143  
-Mt 226, 352, 469, 495

## N

-N 227  
NAMES 694  
NEAR 357, 360, 364, 392, 473  
near 398, 409  
New Project dialog box 30  
New Target dialog box 62  
NO\_ENTRY 377, 484, 681  
NO\_EXIT 379  
NO\_FRAME 381  
NO\_INIT 683  
NO\_INLINE 383  
NO\_LOOP\_UNROLL 384  
NO\_OVERLAP 681  
NO\_PARREGS 385  
NO\_STRING\_CONSTR 387, 439  
-NoBeep 229  
-NoDebugInfo 230  
-NoEnv 232  
-NoPath 233  
NULL 517  
Numbers 417

## O

-Oa 235  
-Ob 236  
-Obfv 236  
Object  
    File 132  
object-file Fformats 77  
-ObjN 238  
OBJPATH 94, 125, 132  
-Oc 240  
-Od 242

---

-OdocF 137, 139, 242, 687  
-Odocf 342  
-Of 244  
offsetof 517  
-Oi 137, 247  
-Oilib 249  
-Ol 252  
-Ona 254  
-OnB 255  
-Onbf 257  
-Onbt 259  
-Onca 261  
ONCE 388  
-Oncn 263  
-OnCopyDown 265  
-OnCstVar 267  
-One 268  
-Onf 244  
-OnP 270  
-OnPMNC 272  
-Ont 273  
-Onu 283  
-OnX 280  
operator  
    # 398  
    ## 398  
    defined 398  
OPTIMIZATION 138, 140  
Optimization  
    Branches 431, 475  
    Lazy Instruction Selection 474  
    Shift optimizations 431  
    Strength Reduction 431, 474  
    Time vs. Size 234, 476  
    Tree Rewriting 432  
OPTION 389  
Option  
    CODE 138  
    CODE GENERATION 140  
    HOST 138, 140  
    INPUT 138, 140  
    LANGUAGE 140  
        LANGUAGE 138  
    MESSAGE 140

MESSAGES 139  
OPTIMIZATION 138, 140  
OUTPUT 138, 140  
STARTUP 139  
TARGET 139  
VARIOUS 139, 140  
Options 68, 69, 70, 71, 72, 73, 705, 718  
-Or 137, 281, 451  
-Os 234, 341, 433  
-Ot 234, 341  
-Ou 283  
OUTPUT 138, 140  
OVERLAP 681

## **P**

P&E 697  
Parsing Recursion 417  
Path List 113  
PC-Lint 61  
PCLint Main Settings preference panel 64  
PCLint Options preference panel 65  
PC-lint Settings preference panel 63  
-Pe 285  
perror 598  
-Pio 287  
piper.exe 59  
PLACEMENT 671  
Pointer  
    \_\_far 404  
    Compatibility 410  
pow 599  
powf 599  
Precedence 702  
Predefined Macro 339  
Premia 89  
Preprocessor  
    Directives 397  
printf 498, 600  
printf.c 498  
Procedure  
    Return Value 470, 471  
    Stack Frame 471  
Processor Expert panel 33  
-Prod 113, 289

---

project directory 37  
project window 38  
project.ini 113, 116, 135  
ptrdiff\_t 342, 516  
putc 601  
putchar 602  
puts 603  
PVCS 128

## Q

qsort 604  
-Qvpt 290

## R

raise 606  
RAM 690  
rand 607  
RAND\_MAX 518  
realloc 497, 608  
RecentCommandLine 715  
Recursive comments 677  
register 397  
Register intitialisation 494  
regservers.bat 61  
remove 609  
rename 610  
Restriction  
    Implementation 415  
return 397  
Return Value 470, 471  
rewind 611  
RGB 305, 306, 307, 308, 309  
ROM 434, 690, 694  
ROM libraries 494  
ROM\_VAR 137, 155, 430  
-Rpe 291  
-Rpt 291

## S

SaveAppearance 706  
SaveEditor 706  
SaveOnExit 706  
SaveOptions 707

scanf 612  
SCHAR\_MAX 512  
SCHAR\_MIN 512  
SEEK\_CUR 517  
SEEK\_END 517  
SEEK\_SET 518  
Segment 473  
    SHORT 473  
Segmentation 427  
Select File to Compile dialog box 49  
Service Name 90  
setbuf 613  
setjmp 614  
setjmp.h 515  
setlocale 615  
setvbuf 616  
Shift optimizations 431  
SHORT 360, 364, 473  
short 397  
SHORT Segments 473  
-ShowAboutDialog 59  
-ShowBurnerDialog 59  
ShowConfigurationDialog 59  
-ShowMessageDialog 59  
-ShowOptionDialog 59  
-ShowSmartSliderDialog 59  
ShowTipOfDay 708  
SHRT\_MAX 513  
SHRT\_MIN 513  
sig\_atomic\_t 515  
SIG\_DFL 516  
SIG\_ERR 516  
SIG\_IGN 516  
SIGABRT 516  
SIGFPE 516  
SIGILL 516  
SIGINT 516  
signal 617  
signal.c 497  
signal.h 515  
Signals 497  
signed 397  
SIGSEGV 516  
SIGTERM 516

---

sin 618  
sinf 618  
single object file 74  
sinh 619  
Size  
    Type 464  
size\_t 342, 516  
sizeof 397  
SKIP1 476  
SKIP2 476  
SMALL memory model 463  
Smart Control 101  
Source File 131  
Special Modifiers 140  
sprintf 620  
sqrt 624  
sqrtf 624  
srand 625  
sscanf 626  
Stack  
    Frame 471  
Stack Frames 471  
STACK\_TEST 392  
Standard Types 98  
start 81  
start08.c 495  
start08.o 495  
start08s.o 495  
start08t.o 495  
start08ts.o 495  
STARTUP 139  
startup 113  
startup command-line options 59  
Startup dialog box 29  
Startup Files 494  
startup option 59  
startup.c 495  
static 397  
StatusBarEnabled 716  
stdarg 410  
stdarg.h 410, 520  
stddef.h 516  
stderr 518  
stdin 518  
stdio.h 517  
stdlib. 498  
stdlib.c 498  
stdlib.h 518, 625  
stdout 335, 518  
strcat 630  
strchr 631  
strcmp 632  
strcoll 633  
strep\_y 249, 634  
strespn 635  
Strength Reduction 431, 474  
strerror 636  
strftime 637  
string.h 519  
STRING\_SECTION 392  
STRING\_SEG 392  
Strings 404  
strlen 249, 639  
strncat 640  
strncmp 641  
strncpy 642  
strpbrk 643  
strrchr 644  
strspn 645  
strstr 646  
strtod 646  
strtok 648  
strtol 649  
strtoul 651  
struct 397  
strxfrm 652  
switch 397  
synchronization 80  
system 653

**T**  
-T 293, 464  
Table 1.13 139  
tan 654  
tanf 654  
tanh 655  
tanhf 655  
TARGET 139

---

Target Settings Preference Panel 61  
Target Settings preference panel 61  
termination 80  
TEST\_CODE 394  
TEXTPATH 94, 126, 199, 214, 221, 222  
time 656  
time.h 519  
time\_t 519  
TINY memory model 463  
Tip of the Day 81  
TipFilePos 708  
TipTimeStamp 708  
TMP 127  
TMP\_MAX 518  
tmpfile 657  
tmpnam 658  
tolower 659  
ToolBarEnabled 716  
Topic Name 90  
toupper 660  
Translation Limits 415  
TRAP\_PROC 396, 412, 472, 678  
Type  
    Floating Point 465  
    Size 464  
Type Declarations 417  
Type Sizes 70  
typedef 397

## U

UCHAR\_MAX 512  
UINT\_MAX 513  
ULONG\_MAX 513  
UltraEdit 90  
ungetc 661  
union 397  
UNIX 110  
unsigned 397  
Use custom PRM file 72  
Use Decoder to generate Disassembly Listing 68,  
    70  
Use template PRM file 72  
Use third party debugger 66  
USELIBPATH 128

USERNAME 129  
USHRT\_MAX 513

## V

-V 299  
va\_arg 410, 662  
va\_end 662  
va\_start 662  
VARIOUS 139, 140  
VECTOR 412  
vfprintf 663  
-View 300  
Visual C++ 74  
void 397  
volatile 397, 425  
vprintf 663  
vsprintf 498, 663

## W

-W1 337  
-W2 338, 674  
wchar\_t 342, 516  
wcstombs 498, 665  
wctomb 498, 664  
-WErrFile 302  
while 397  
WindowFont 717  
WindowPos 717  
Windows 110  
Winedit 89  
Wizard 29  
Wizard Map 29  
-Wmsg8x3 304  
-WmsgCE 305  
-WmsgCF 306  
-WmsgCI 307  
-WmsgCU 308  
-WmsgCW 309  
-WmsgFb 304, 310, 313, 315, 317, 319, 321  
-WmsgFbi 310  
-WmsgFbm 310  
-WmsgFi 304, 312, 317, 319, 321  
-WmsgFim 312  
-WmsgFiv 312

---

- WmsgFob 314, 317
- WmsgFoi 315, 316, 319, 321
- WmsgFonf 318
- WmsgFonp 315, 317, 319, 320, 321
- WmsgNe 322
- WmsgNi 323
- WmsgNu 324
- WmsgNw 326
- WmsgSd 327
- WmsgSe 328
- WmsgSi 329
- WmsgSw 330
- WOutFile 331
- Wpd 333
- WStdout 335

## **Z**

- Zero Out 494
- zero out 404
- Zero Page 473