



RS08 Build Tools Reference Manual for Microcontrollers

Revised: 18 October 2007





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 © 2007 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

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

Table of Contents

I Overview

II Using the Compiler

1	Introduction	27
	Compiler Environment	27
	Project Directory	28
	Editor	28
	Using the CodeWarrior IDE to Manage a Project	28
	New Project Wizard	29
	Analysis of the Project Files and Folders	37
	Compilation	46
	Linking with the Linker	58
	Using the Simulator/Debugger	59
	Application Programs (Build Tools)	61
	Startup Command-Line Options	61
	Highlights	62
	CodeWarrior Integration	63
	Combined or Separated Installations	63
	Target Settings Preference Panel	64
	Build Extras Preference Panel	65
	Assembler for RS08 Preference Panel	67
	Burner Preference Panel	68
	Compiler for RS08 Preference Panel	69
	Importer for RS08 Preference Panel	70
	Linker for RS08 Preference Panel	71
	CodeWarrior Tips and Tricks	72
	Integration into Microsoft Visual Studio (Visual C++ V5.0 or later)	73
	Object-File Formats	75

Table of Contents

ELF/DWARF Object-File Format	75
Tools	76
Mixing Object-File Formats	76
2 Graphical User Interface	77
Launching the Compiler	78
Interactive Mode	78
Batch Mode	78
Tip of the Day	79
Main Window	80
Window Title	81
Content Area	82
Toolbar	83
Status Bar	84
Menu Bar	84
File Menu	84
Editor Settings dialog box	86
Save Configuration dialog box	93
Environment Configuration Dialog Box	94
Compiler Menu	96
View Menu	97
Help Menu	98
Standard Types dialog box	98
Option Settings dialog box	100
Compiler Smart Control dialog box	102
Message Settings dialog box	104
Changing the Class associated with a Message	106
Retrieving Information about an Error Message	106
About dialog box	107
Specifying the Input File	107
Use the Command Line in the Toolbar to Compile	107
Message/Error Feedback	108
Use Information from the Compiler Window	109
Use a User-Defined Editor	109

3	Environment	111
	Current Directory	112
	Environment Macros	113
	Global Initialization File (mcutools.ini)	114
	Local Configuration File (usually project.ini)	114
	Paths	115
	Line Continuation	116
	Environment Variable Details	117
	COMPOPTIONS: Default Compiler Options	117
	COPYRIGHT: Copyright entry in object file	118
	DEFAULTDIR: Default Current Directory	119
	ENVIRONMENT: Environment File Specification	120
	ERRORFILE: Error filename Specification	121
	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
4	Files	131
	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
5	Compiler Options	135
	Option Recommendation	137

Table of Contents

Compiler Option Details	138
Option Groups	138
Option Scopes	139
Option Detail Description	140
-!: filenames to DOS length	142
-AddIncl: Additional Include File	143
-Ansi: Strict ANSI	144
-BfaB: Bitfield Byte Allocation	145
-BfaGapLimitBits: Bitfield Gap Limit	147
-BfaTSR: Bitfield Type-Size Reduction	148
-C++ (-C++f, -C++e, -C++c): C++ Support	150
-Cc: Allocate Constant Objects into ROM	151
-Ccx: Cosmic Compatibility Mode for Space Modifiers and Interrupt Handlers	153
-Ci: Tri- and Bigraph Support	155
-Cn: Disable compactC++ features	159
-Cni: No Integral Promotion	160
-Cpcc: C++ Comments in ANSI-C	162
-Cq: Propagate const and volatile qualifiers for structs	164
-CswMaxLF: Maximum Load Factor for Switch Tables	165
-CswMinLB: Minimum Number of Labels for Switch Tables	167
-CswMinLF: Minimum Load Factor for Switch Tables	168
-CswMinSLB: Minimum Number of Labels for Search Switch Tables	170
-Cu: Loop Unrolling	171
-Cx: No Code Generation	174
-D: Macro Definition	174
-Ec: Conversion from 'const T*' to 'T*'	176
-Eencrypt: Encrypt Files	178
-Ekey: Encryption Key	179
-Env: Set Environment Variable	180
-F (-F2, -F2o): Object-File Format	181
-H: Short Help	182
-I: Include File Path	183
-La: Generate Assembler Include File	184
-Lasm: Generate Listing File	186

-Lasmc: Configure Listing File	187
-Ldf: Log Predefined Defines to File	188
-Li: List of Included Files	190
-Lic: License Information	191
-LicA: License Information about every Feature in Directory	192
-LicBorrow: Borrow License Feature	193
-LicWait: Wait until Floating License is Available from Floating License Server	195
-Ll: Statistics about Each Function	196
-Lm: List of Included Files in Make Format.	197
-LmCfg: Configuration of List of Included Files in Make Format	199
-Lo: Object File List	201
-Lp: Preprocessor Output	202
-LpCfg: Preprocessor Output configuration	203
-LpX: Stop after Preprocessor.	205
-N: Display Notify Box	206
-NoBeep: No Beep in Case of an Error.	207
-NoDebugInfo: Do not Generate Debug Information	208
-NoPath: Strip Path Info	209
-Oa: Alias Analysis Options	210
-O (-Os, -Ot): Main Optimization Target	211
-ObjN: Object filename Specification.	212
-Obsr: Generate Always Near Calls	213
-Od: Disable Mid-Level Optimizations.	215
-Odb: Disable Mid-Level Branch Optimizations	216
-OdocF: Dynamic Option Configuration for Functions	218
-Oi: Inlining.	220
-Oilib: Optimize Library Functions.	221
-OnB: Disable Branch Optimizer	223
-OnBRA: Disable JAL to BRA Optimization	224
-Onbsr: Disable far to near call optimization	228
-OnCopyDown: Do Generate Copy Down Information for Zero Values.	230
-OnCstVar: Disable CONST Variable by Constant Replacement.	231
-Onp: Disable Peephole Optimizer	232

Table of Contents

-OnPMNC: Disable Code Generation for NULL Pointer to Member Check	233
-Onr: Disable Reload from Register Optimization	234
-Ont: Disable Tree Optimizer	235
-Ontc: disable tail call optimization.	241
-Ostk: Reuse Locals of Stack Frame	243
-Pe: Preprocessing Escape Sequences in Strings.	244
-Pio: Include Files Only Once	246
-Prod: Specify Project File at Startup	247
-Qvtp: Qualifier for Virtual Table Pointers	249
-T: Flexible Type Management	250
-V: Prints the Compiler Version.	256
-View: Application Standard Occurrence	257
-WErrFile: Create "err.log" Error File.	258
-Wmsg8x3: Cut filenames in Microsoft Format to 8.3	259
-WmsgCE: RGB Color for Error Messages	260
-WmsgCF: RGB Color for Fatal Messages.	261
-WmsgCI: RGB Color for Information Messages.	262
-WmsgCU: RGB Color for User Messages.	263
-WmsgCW: RGB Color for Warning Messages	264
-WmsgFb (-WmsgFbi, -WmsgFbm): Set Message File Format for Batch Mode	264
-WmsgFi (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode	266
-WmsgFob: Message Format for Batch Mode	268
-WmsgFoi: Message Format for Interactive Mode	270
-WmsgFonf: Message Format for no File Information	273
-WmsgFonp: Message Format for no Position Information	275
-WmsgNe: Number of Error Messages	276
-WmsgNi: Number of Information Messages	277
-WmsgNu: Disable User Messages	278
-WmsgNw: Number of Warning Messages.	280
-WmsgSd: Setting a Message to Disable.	281
-WmsgSe: Setting a Message to Error.	282
-WmsgSi: Setting a Message to Information.	283

-WmsgSw: Setting a Message to Warning	284
-WOutFile: Create Error Listing File	285
-Wpd: Error for Implicit Parameter Declaration	286
-WStdout: Write to Standard Output.	287
-W1: No Information Messages	288
-W2: No Information and Warning Messages.	289
6 Compiler Predefined Macros	291
Compiler Vendor Defines	292
Product Defines.	292
Data Allocation Defines	292
Various Defines for Compiler Option Settings.	293
Option Checking in C Code	294
ANSI-C Standard Types 'size_t', 'wchar_t' and 'ptrdiff_t' Defines	295
Macros for RS08	297
Division and Modulus.	297
Macros for RS08	298
Object-File Format Defines	298
Bitfield Defines	298
Bitfield Allocation.	298
Bitfield Type Reduction	300
Sign of Plain Bitfields	301
Type Information Defines	302
Freescale RS08-Specific Defines	305
7 Compiler Pragmas	307
Pragma Details	307
#pragma CONST_SEG: Constant Data Segment Definition	309
#pragma CREATE_ASM_LISTING: Create an Assembler Include File Listing	312
#pragma DATA_SEG: Data Segment Definition	313
#pragma INLINE: Inline Next Function Definition	315
#pragma INTO_ROM: Put Next Variable Definition into ROM	316
#pragma LINK_INFO: Pass Information to the Linker	318
#pragma LOOP_UNROLL: Force Loop Unrolling	319

Table of Contents

#pragma mark: Entry in CodeWarrior IDE Function List	320
#pragma MESSAGE: Message Setting	322
#pragma NO_ENTRY: No Entry Code	323
#pragma NO_EXIT: No Exit Code	325
#pragma NO_FRAME: No Frame Code	327
#pragma NO_INLINE: Do not Inline next function definition	328
#pragma NO_LOOP_UNROLL: Disable Loop Unrolling	329
#pragma NO_RETURN: No Return Instruction	330
#pragma NO_STRING_CONSTR: No String Concatenation during preprocessing	332
#pragma ONCE: Include Once	333
#pragma OPTION: Additional Options.	334
#pragma REALLOC_OBJ: Object Reallocation.	336
#pragma STRING_SEG: String Segment Definition	338
#pragma TEST_CODE: Check Generated Code.	340
#pragma TRAP_PROC: Mark function as interrupt function	342

8 ANSI-C Frontend 343

Implementation Features	343
Keywords	344
Preprocessor Directives	344
Language Extensions	344
Implementation-Defined Behavior	360
Translation Limitations	361
ANSI-C Standard	365
Integral Promotions	365
Signed and Unsigned Integers	365
Arithmetic Conversions	365
Order of Operand Evaluation.	366
Rules for Standard-Type Sizes.	367
Floating-Type Formats	367
Floating-Point Representation of 500.0 for IEEE	367
Representation of 500.0 in IEEE32 Format.	368
Volatile Objects and Absolute Variables	369
Bitfields	369

Signed Bitfields	370
Segmentation	371
Example of Segmentation without the -Cc Compiler Option	373
Example of Segmentation with the -Cc Compiler Option	374
Optimizations	374
Peephole Optimizer	374
Strength Reduction	376
Shift Optimizations	376
Branch Optimizations	376
Dead-Code Elimination	376
Constant-Variable Optimization	376
Tree Rewriting	377
Using Qualifiers for Pointers	379
Defining C Macros Containing HLI Assembler Code	382
Defining a Macro	382
Using Macro Parameters	384
Using the Immediate-Addressing Mode in HLI Assembler Macros	384
Generating Unique Labels in HLI Assembler Macros	385
Generating Assembler Include Files (-La Compiler Option)	386
9 Generating Compact Code	397
Compiler Options	397
-Oi: Inline Functions	397
Relocatable Data	398
Using -Ostk	399
Programming Guidelines	399
Constant Function at a Specific Address	400
HLI Assembly	400
Post- and Pre-Operators in Complex Expressions	401
Boolean Types	402
printf() and scanf()	402
Bitfields	402
Struct Returns	403
Local Variables	404
Parameter Passing	404

Table of Contents

Unsigned Data Types	404
Inlining and Macros	405
Data Types	406
Tiny or Short Segments	406
Qualifiers	406
10 RS08 Backend	407
Non-ANSI Keywords	407
Data Types	407
Scalar Types	407
Floating Point Types	409
Pointer Types and Function Pointers	409
Structured Types, Alignment	409
Bit Fields	409
Register Usage	410
Parameter Passing	410
Entry and Exit Code	411
Pragmas	411
TRAP_PROC	411
NO_ENTRY	412
NO_EXIT	412
Interrupt Functions	412
#pragma TRAP_PROC	412
Interrupt Vector Table Allocation	412
Segmentation	412
Optimizations	413
Lazy Instruction Selection	413
Branch Optimizations	413
Constant Folding	413
Volatile Objects	413
Programming Hints	414
11 High-Level Inline Assembler for the Freescale RS08	415
Syntax	415
Mixing HLI Assembly and HLL	416

Example	416
C Macros	416
Special Features	417
Caller/Callee Saved Registers	417
Reserved Words	418
__asm MOV #%HIGH_6_13(var), __PAGESEL	418
Pseudo-Opcodes	418
Accessing Variables	419
Constant Expressions	419

III ANSI-C Library Reference

12 Library Files 423

Directory Structure	423
Generating a Library	423
Common Source Files	424
Startup Files	425
Library Files	425

13 Special Features 427

Memory Management -- malloc(), free(), calloc(), realloc(); alloc.c, and heap.c	427
Signals - signal.c	427
Multi-byte Characters - mblen(), mbtowc(), wctomb(), mbstowcs(), westombs(); stdlib.c	428
Program Termination - abort(), exit(), atexit(); stdlib.c	428
I/O - printf.c	428
Locales - locale.*	430
ctype	430
String Conversions - strtol(), strtoul(), strtod(), and stdlib.c	430

14 Library Structure 431

Error Handling	431
--------------------------	-----

Table of Contents

String Handling Functions	431
Memory Block Functions	432
Mathematical Functions	432
Memory Management	434
Searching and Sorting	435
System Functions	436
Time Functions	436
Locale Functions	437
Conversion Functions	437
printf() and scanf()	438
File I/O	438

15 Types and Macros in the Standard Library 441

errno.h	441
float.h	441
limits.h	443
locale.h	444
math.h	446
setjmp.h	446
signal.h	446
stddef.h	447
stdio.h	447
stdlib.h	448
time.h	449
string.h	449
assert.h	450
stdarg.h	450
ctype.h	451

16 The Standard Functions 453

abort()	454
abs()	454
acos() and acosf()	455
asctime()	456
asin() and asinf()	456

assert()	457
atan() and atanf()	457
atan2() and atan2f()	458
atexit()	459
atof()	459
atoi()	460
atol()	461
bsearch()	461
calloc()	463
ceil() and ceilf()	463
clearerr()	464
clock()	464
cos() and cosf()	465
cosh() and coshf()	465
ctime()	466
difftime()	466
div()	467
exit()	467
exp() and expf()	468
fabs() and fabsf()	469
fclose()	469
feof()	470
ferror()	470
fflush()	471
fgetc()	471
fgetpos()	472
fgets()	473
floor() and floorf()	473
fmod() and fmodf()	474
fopen()	474
fprintf()	476
fputc()	476
fputs()	477
fread()	477
free()	478

Table of Contents

freopen()	478
frexp() and frexpf()	479
fscanf()	479
fseek()	480
fsetpos()	481
ftell()	481
fwrite()	482
getc()	482
getchar()	483
getenv()	483
gets()	483
gmtime()	484
isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), and isxdigit()	485
labs()	486
ldexp() and ldexpf()	486
ldiv()	487
localeconv()	487
localtime()	488
log() and logf()	488
log10() and log10f()	489
longjmp()	490
malloc()	490
mblen()	491
mbstowcs()	491
mbtowc()	492
memchr()	492
memcmp()	493
memcpy() and memmove()	494
memset()	494
mktime()	495
modf() and modff()	495
perror()	496
pow() and powf()	496
printf()	497

putc()	498
putchar()	498
puts()	498
qsort()	499
raise()	500
rand()	501
realloc()	501
remove()	502
rename()	502
rewind()	503
scanf()	503
setbuf()	504
setjmp()	504
setlocale()	505
setvbuf()	506
signal()	507
sin() and sinf()	508
sinh() and sinhf()	508
sprintf()	509
sqrt() and sqrtf()	513
srand()	513
sscanf()	514
strcat()	518
strchr()	518
strcmp()	519
strcoll()	519
strcpy()	520
strcspn()	520
strerror()	521
strftime()	522
strlen()	523
strncat()	524
strncmp()	524
strncpy()	525
strpbrk()	525

Table of Contents

strchr()	526
strspn()	526
strstr()	527
strtod()	528
strtok()	528
strtol()	529
strtoul()	530
strxfrm()	531
system()	532
tan() and tanf()	532
tanh() and tanhf()	533
time()	534
tmpfile()	534
tmpnam()	535
tolower()	535
toupper()	536
ungetc()	536
va_arg(), va_end(), and va_start()	537
vfprintf(), vprintf(), and vsprintf()	538
wctomb()	538
wctombs()	539

IV Appendices

A Porting Tips and FAQs **543**

Migration Hints	543
Porting from Cosmic	543
Allocation of Bitfields	549
Type Sizes and Sign of char	549
@bool Qualifier	550
@tiny and @far Qualifier for Variables	550
Arrays with Unknown Size	550
Missing Prototype	551

_asm(“sequence”)	551
Recursive Comments.	551
Interrupt Function, @interrupt	551
Defining Interrupt Functions.	552
General Optimization Hints	555
Frequently Asked Questions (FAQs), Troubleshooting	556
Making Applications.	556
EBNF Notation	561
Terminal Symbols	562
Non-Terminal Symbols	562
Vertical Bar	562
Brackets.	563
Parentheses	563
Production End	563
EBNF Syntax.	563
Extensions	564
Abbreviations, Lexical Conventions.	564
Number Formats	565
Precedence and Associativity of Operators for ANSI-C	566
List of all Escape Sequences.	568
B Global Configuration File Entries	569
[Options] Section	569
DefaultDir	569
[XXX_Compiler] Section	570
SaveOnExit	570
SaveAppearance	570
SaveEditor	570
SaveOptions.	571
RecentProject0, RecentProject1, etc.	571
TipFilePos	572
ShowTipOfDay	572
TipTimeStamp.	572
[Editor] Section.	573
Editor_Name	573

Table of Contents

Editor_Exe	573
Editor_Opts	574
Example.	574
C Local Configuration File Entries	577
[Editor] Section	577
Editor_Name	577
Editor_Exe	578
Editor_Opts	578
Example [Editor] Section	578
[XXX_Compiler] Section	579
RecentCommandLineX	579
CurrentCommandLine	579
StatusBarEnabled	580
ToolbarEnabled	580
WindowPos	581
WindowFont	581
Options.	582
EditorType	582
EditorCommandLine	583
EditorDDEClientName	583
EditorDDETopicName	583
EditorDDEServiceName	584
Example.	584
D Known C++ Issues in the RS08 Compilers	587
Template Issues	587
Operators	588
Binary Operators	589
Unary operators	590
Equality Operators.	591
Header Files.	592
Bigraph and Trigraph Support.	592
Known Class Issues.	593
Keyword Support.	595

Member Issues	595
Constructor and Destructor Functions	598
Overload Features	601
Conversion Features	603
Standard Conversion Sequences	603
Ranking implicit conversion sequences	604
Explicit Type Conversion	605
Initialization Features	606
Errors	608
Other Features.	610



Table of Contents

Overview

The RS08 Build Tools Reference Manual for Microcontrollers describes the ANSI-C/C++ 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 of 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 CodeWarrior Development Studio and 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.
- [RS08 Backend](#): Description of code generator and basic type implementation, also information about hardware-oriented programming (optimizations, interrupt functions, etc.) specific for the Freescale RS08.
- [High-Level Inline Assembler for the Freescale RS08](#): Description of the HLI Assembler for the RS08.

Introduction

This chapter describes the Compiler used for the Freescale RS08. The Compiler consists of a **Frontend**, which is language-dependent, and a **Backend** that depends on the target processor, the RS08.

The major sections of this chapter are:

- [Compiler Environment](#)
- [Using the CodeWarrior IDE to Manage a Project](#)
- [Compilation](#)
- [Application Programs \(Build Tools\)](#)
- [Startup Command-Line Options](#)
- [Highlights](#)
- [CodeWarrior Integration](#)
- [Integration into Microsoft Visual Studio \(Visual C++ V5.0 or later\)](#)
- [Object-File Formats](#)

Compiler Environment

You can use the Compiler as a transparent, integral part of the CodeWarrior Development Studio. Using the CodeWarrior IDE is the recommended way to get your project up and running in minimal time. Alternatively, you can configure the Compiler and use it as a standalone application as a member of a suite of other Build Tool Utilities such as a Linker, Assembler, EPROM Burner, Simulator or Debugger.

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 RS08 Compiler does not directly generate absolute (*.abs) files, but uses the ELF/DWARF object file format to produce relocatable object code. Use the CodeWarrior ELF linker to link object files. The Burner uses the resulting absolute files to produce S-Record (*.s19 or *.sx) files for programming ROM memory.

A typical Compiler configuration is associated with a [Project Directory](#) and an [Editor](#).

Introduction

Using the CodeWarrior IDE to Manage a Project

Project Directory

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

When designing a project, you can 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; you can let the CodeWarrior software coordinate and manage the entire project; or, you can begin the construction of your project with the CodeWarrior IDE and use the standalone build tools (Assembler, Compiler, Linker, Simulator/Debugger) 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 Microcontrollers
V6.1\prog.

Editor

You can associate an editor, including the editor that is integrated into CodeWarrior Development Suite, with the Compiler to enable both error or positive feedback. You can use the *Configuration* dialog box to configure the Compiler to select your choice of editors when using the Build Tools. Refer to the [Editor Settings dialog box](#).

Using the CodeWarrior IDE to Manage a Project

The CodeWarrior Development Suite has a New Project 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 placed in the project directory.

Use the information in the [New Project Wizard](#) section to construct and configure a basic CodeWarrior project that uses C source code.

New Project Wizard

Start the RS08 CodeWarrior IDE (usual path: Freescale\CodeWarrior for Microcontrollers V6.1\bin\IDE.exe). The Startup dialog box appears automatically ([Figure 1.1](#)). Alternatively, you can select *File > New Project* to create a project after closing the Startup dialog box. Now, create a new project by following these steps:

Figure 1.1 Startup Dialog Box

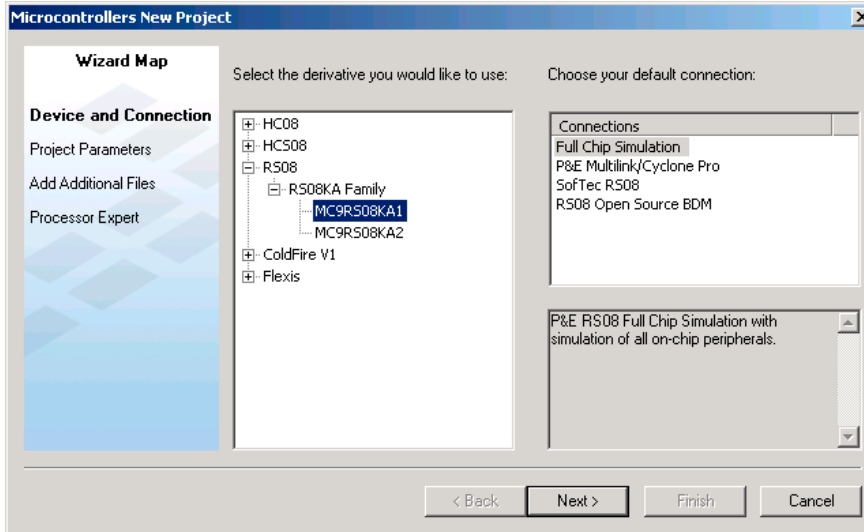


Introduction

Using the CodeWarrior IDE to Manage a Project

1. Select *Create New Project*. The *Microcontrollers New Project Device and Connection* window appears ([Figure 1.2](#)).

Figure 1.2 New Project Device and Connection Dialog Box

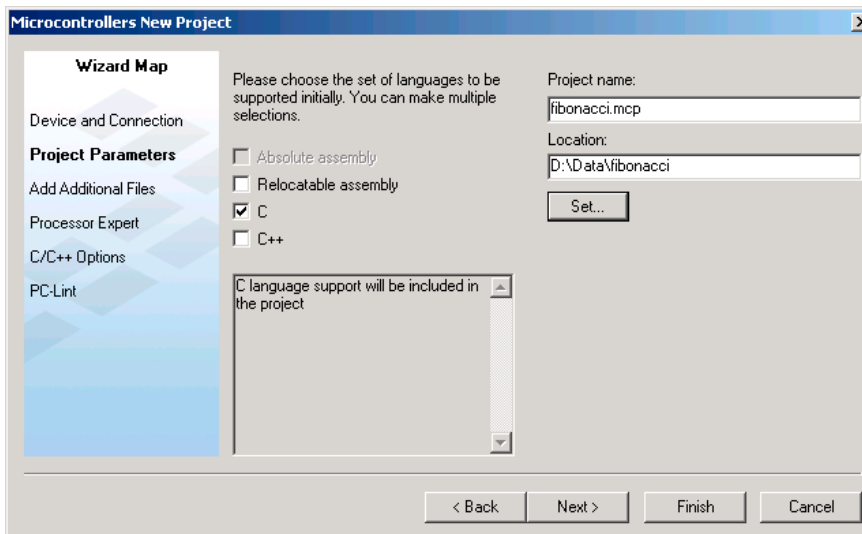


2. Select the RS08 derivative to use for your project.

3. Press *Next* >.

The *Project Parameters* dialog box appears ([Figure 1.3](#)).

Figure 1.3 New Project Wizard - Project Parameters Dialog Box



4. Select the language set for the project.

In this case, select *C*, the default, but leave *Relocatable Assembly* and *C++* unselected. Enter the name for your project in the *Project Name* text box. The CodeWarrior IDE uses the default **.mcp* extension automatically, so you do not have to explicitly append the extension to the filename.

If the default location in the *Location* textbox is not where you want to place the project directory, press the *Set* button to the right of the *Location* textbox and browse to the location of your choice in the *Choose Project Location* dialog box. The wizard automatically creates a folder with the same name as the project.

5. Press the *Save* button to close the *Choose Project Location* dialog box.

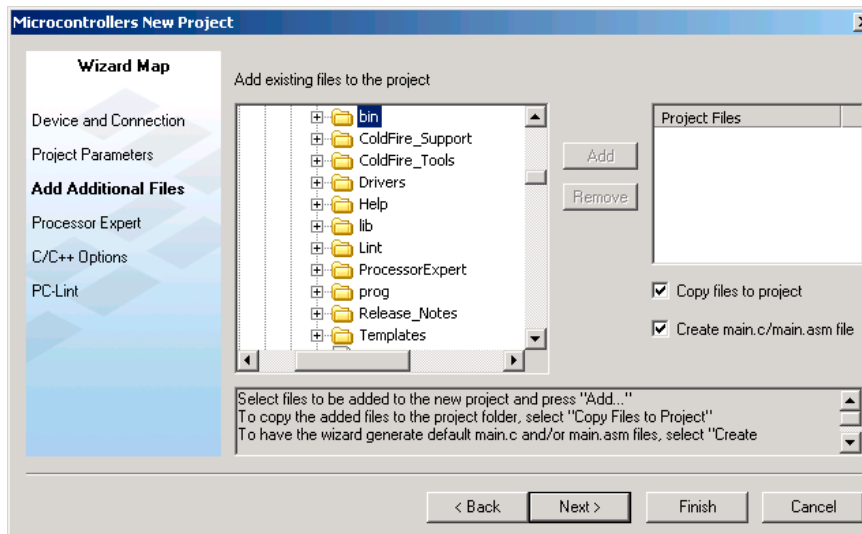
Introduction

Using the CodeWarrior IDE to Manage a Project

6. Press *Next >*.

The *Add Additional Files* dialog box appears ([Figure 1.4](#)).

Figure 1.4 Add Additional Files Dialog Box

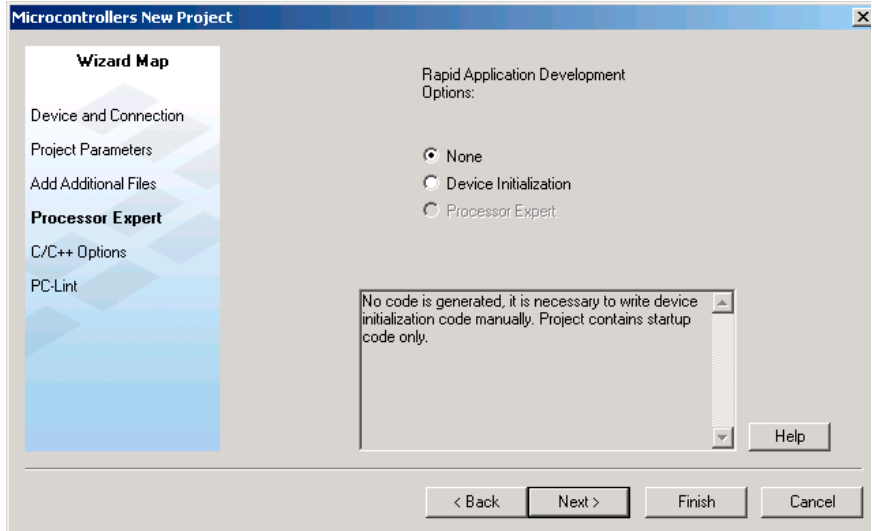


Sample files can be found in the (CodeWarrior Examples) folder in the installation folder, but in this case do not add any files.

7. Click *Next >*.

The *Processor Expert* dialog box appears ([Figure 1.5](#)).

Figure 1.5 Processor Expert Dialog Box



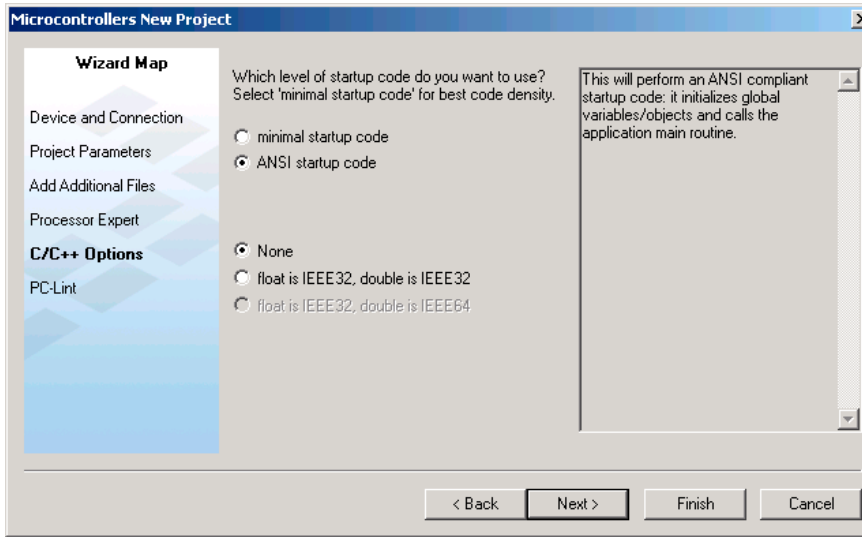
The default, *None*, is already checked. Press *Next >*.

Introduction

Using the CodeWarrior IDE to Manage a Project

The C/C++ *Options* dialog box appears ([Figure 1.6](#)).

Figure 1.6 C/C++ Options Dialog Box



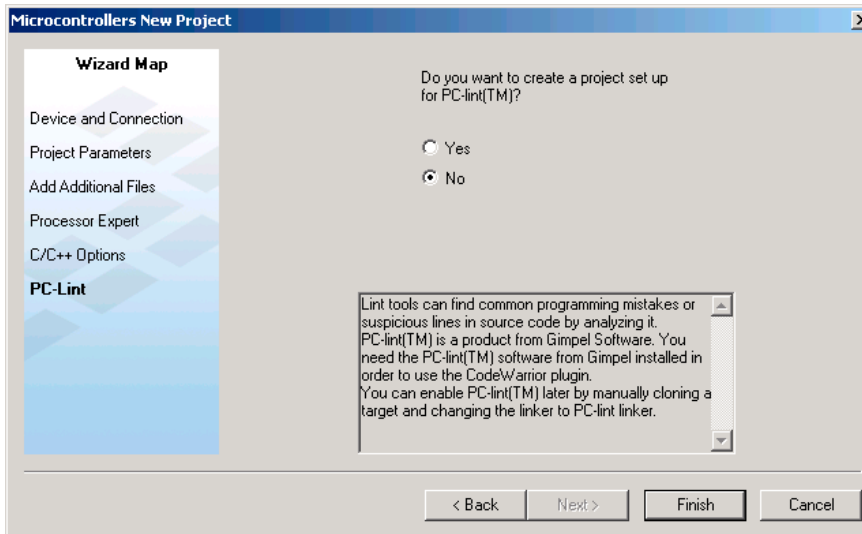
8. Select *ANSI startup code* and *None* for float options.

A simple project does not require the complexity of floating-point numbers. Use the integer format whenever possible in your projects, as floating-point numbers impose a severe speed penalty.

9. Press *Next* >.

The *PCLint* dialog box appears ([Figure 1.7](#)).

Figure 1.7 PCLint dialog box



10. Select *No*.

PCLint is a useful software package for detecting software errors, but is not needed for this project.

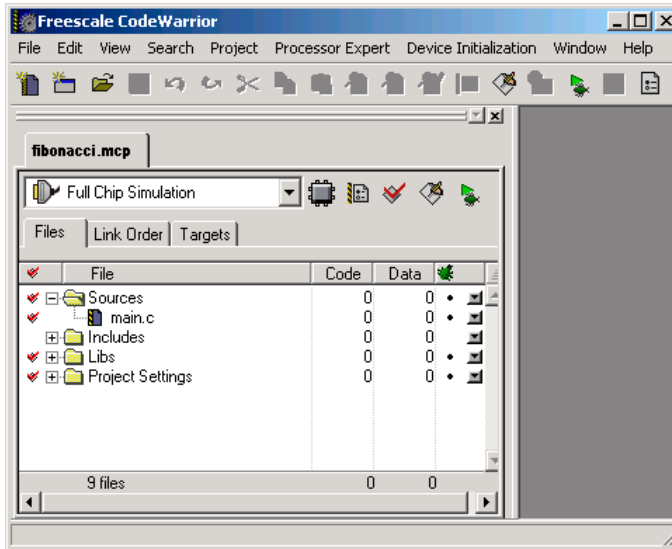
Introduction

Using the CodeWarrior IDE to Manage a Project

11. Press *Finish* >.

The CodeWarrior software now creates an ANSI-C project ([Figure 1.8](#)).

Figure 1.8 CodeWarrior Project Window

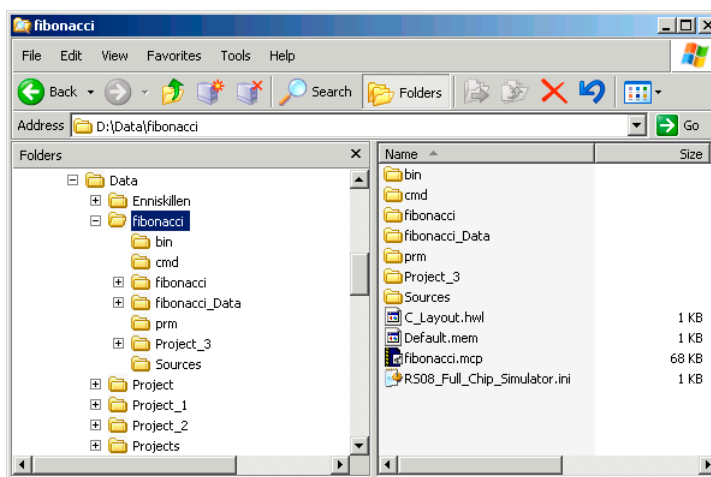


Some files and folders are automatically generated. The root folder is the *project directory* that you selected in the first step.

Analysis of the Project Files and Folders

The CodeWarrior software created a project window that contains two text files and seven folders. The folder icons do not necessarily represent any actual folders but instead are convenient groups of project files. In Windows Explorer, if you examine the project directory that the software created for the project, you can view the actual project folders and files generated, as in [Figure 1.9](#). After the final stage of the New Project Wizard, you can safely close the project and return to it later, in the same configuration as when you last saved it.

Figure 1.9 Project Directory in the Windows Explorer



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

D:\Data\fibonacci

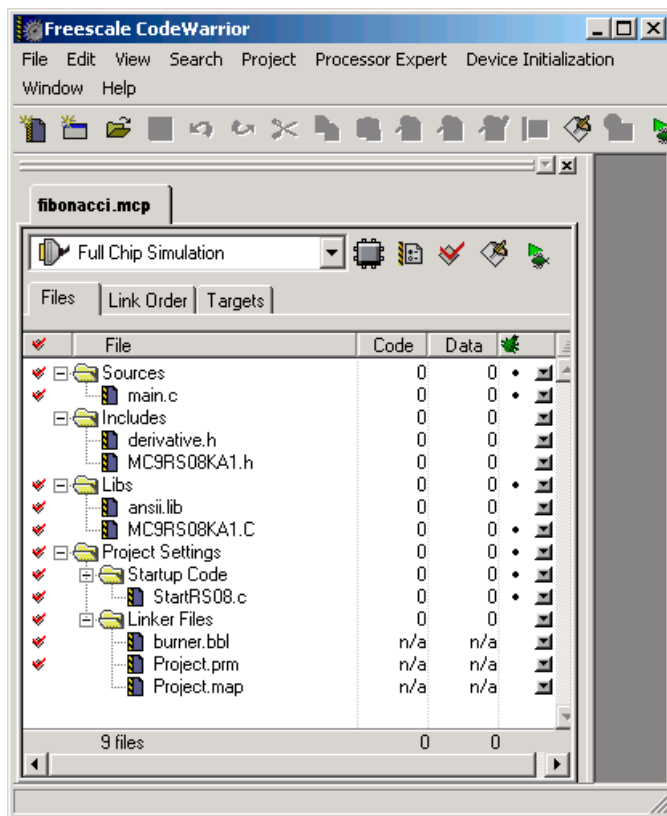
Inside the project directory is the master file for the project `fibonacci.mcp`. This is the file that you open when you want to reopen the project. Opening this master project file opens the CodeWarrior project in the same configuration it had when it was last saved.

Introduction

Using the CodeWarrior IDE to Manage a Project

If you expand the folders (groups) in the CodeWarrior project window, you can view the default files that the CodeWarrior software generated ([Figure 1.10](#)).

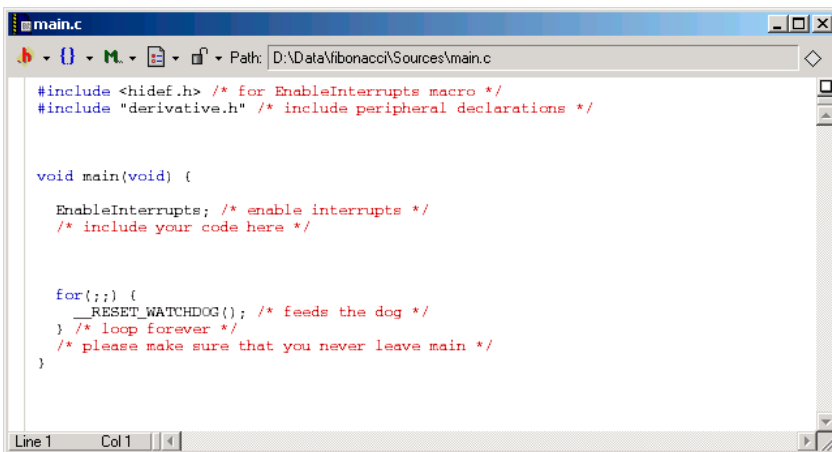
Figure 1.10 Project Window Showing the Files Created by the CodeWarrior IDE



Files marked by red check marks will remain checked until they are successfully assembled, compiled, or linked.

Double click on the `main.c` file in the `Sources` group. The CodeWarrior editor opens the `main.c` file in the project window ([Figure 1.11](#)).

Figure 1.11 main.c Opened in the Project Window



```

main.c
Path: D:\Data\Fibonacci\Sources\main.c

#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 main */
}
    
```

You can adapt the `main.c` file created by the Wizard as a base for your C source program, or you can import other C source-code files into the project and remove the default `main.c` file from the project. Either way, you need only one `main()` function for your project.

For now, use the simple `main.c` file. At this point, the CodeWarrior IDE has created the project, but the source files have not yet been compiled and no object code has been linked into an executable output file. Return to the CodeWarrior project window.

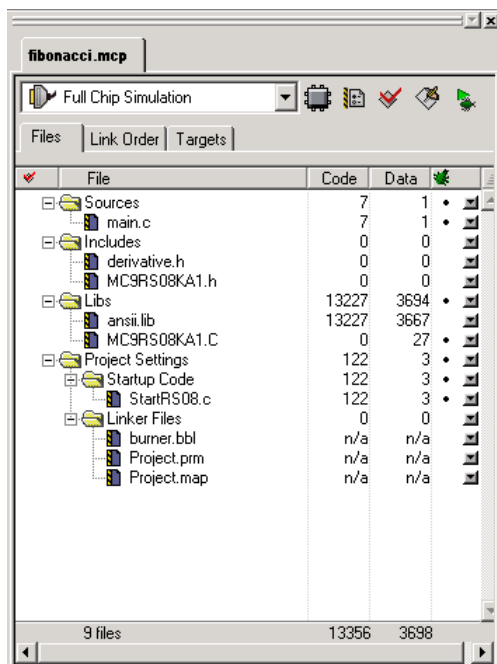
You can process any of the check-marked files individually or a combination of them simultaneously by selecting their icons in the project window. To build the entire project all at once, press the *Make* button on the Toolbar in the project window, or build your project using *Project > Make* or *Project > Debug*.

If the CodeWarrior IDE is correctly configured and the files have no serious errors, all of the red check marks in the project window disappear after a successful build of the project ([Figure 1.12](#)).

Introduction

Using the CodeWarrior IDE to Manage a Project

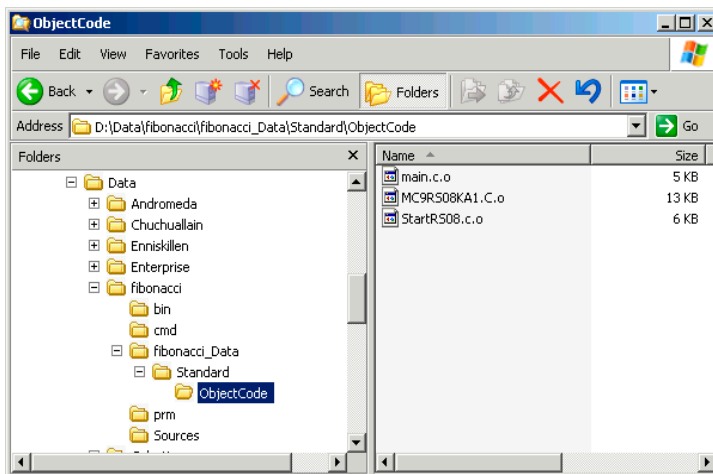
Figure 1.12 Successful Build of your Project



Continually compiling and linking your project files during the construction phase of the project is a wise programming technique in case an error occurs. The source of the error is much easier to locate if the project is frequently rebuilt.

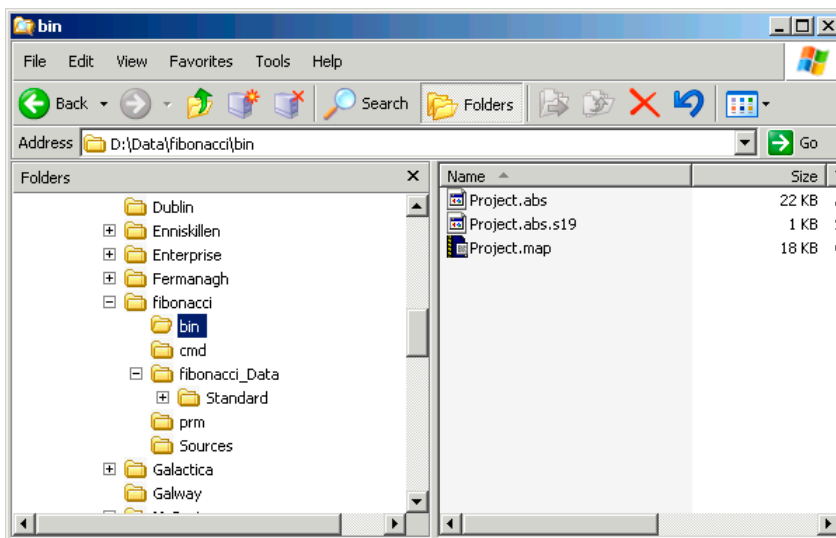
This project has some C source files that successfully compiled. The *Code* and *Data* columns in the project window show the size of the compiled executable object code and the non-executable data in the object code for the compiled source files. Some additional files were generated after the build process ([Figure 1.13](#)).

Figure 1.13 Additional Files after a Project Build



The object-code files for the C-source files are found in the *ObjectCode* folder. However, the executable output file is located in the *bin* folder ([Figure 1.14](#)).

Figure 1.14 bin Folder in the Project Directory



Introduction

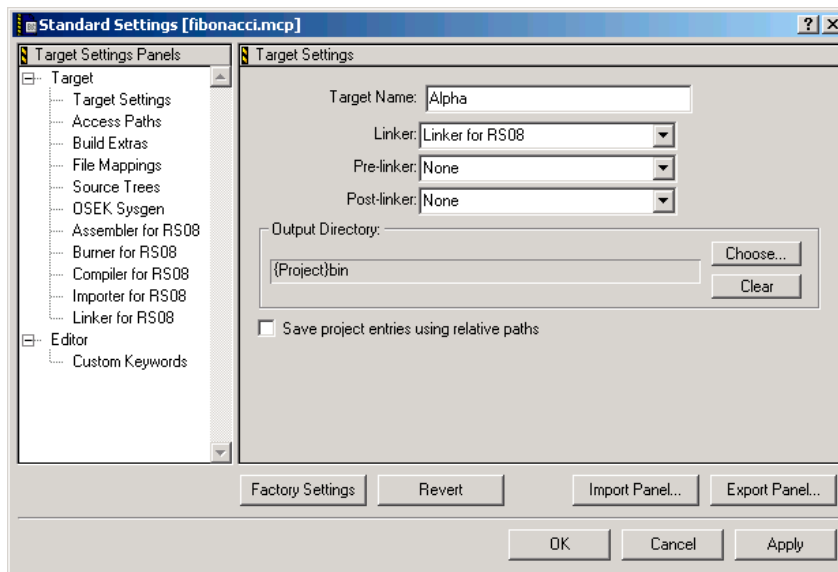
Using the CodeWarrior IDE to Manage a Project

All the files currently in the `bin` folder have the `Project` filename plus an extension. The extension for the executable is `*.abs` (for absolute). The `*.s19` file extension is the *S-Record File* used for programming ROM memory. The `*.map` file extension is for the *Linker Map file*. The Map file provides (among other things) useful information concerning how the Linker allocates RAM and ROM memory areas for the various modules used in the project.

You did not enter these filenames when creating the project with the Project Wizard. These are the default filenames for the project when using the New Project Wizard. You can change these defaults to be more meaningful, say *Alpha.**, by using the *Target Settings* preference panels available in the CodeWarrior IDE.

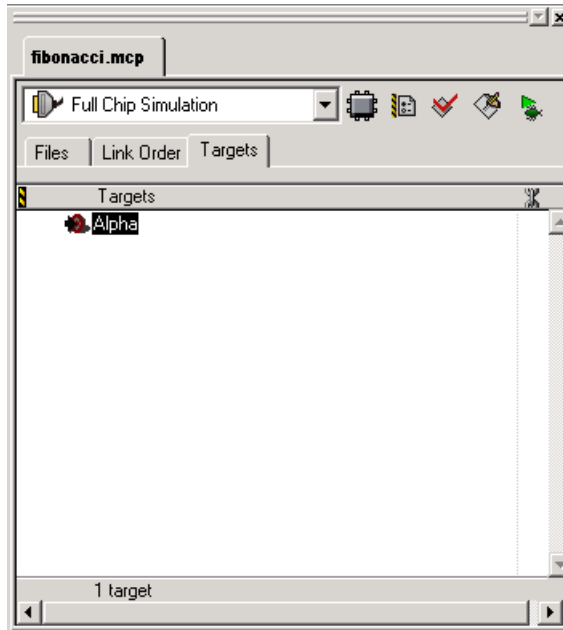
From the *Edit* menu in the CodeWarrior IDE, select *Edit > Standard Settings*. The *Target Settings* dialog box appears with the *Target Settings* preference panel ([Figure 1.15](#)).

Figure 1.15 Target Settings Preference Panel



The *Target Name*: text box contains the default *Target Name* for the project. Enter `Alpha` in this text box and press *OK*. Select the *Edit* menu to see that the *Standard Settings* menu item has been replaced by *Alpha Settings*. This change is also reflected in the project window. *Alpha* now appears as the new name for the build target ([Figure 1.16](#)).

Figure 1.16 Alpha is the New Name for the Build Target

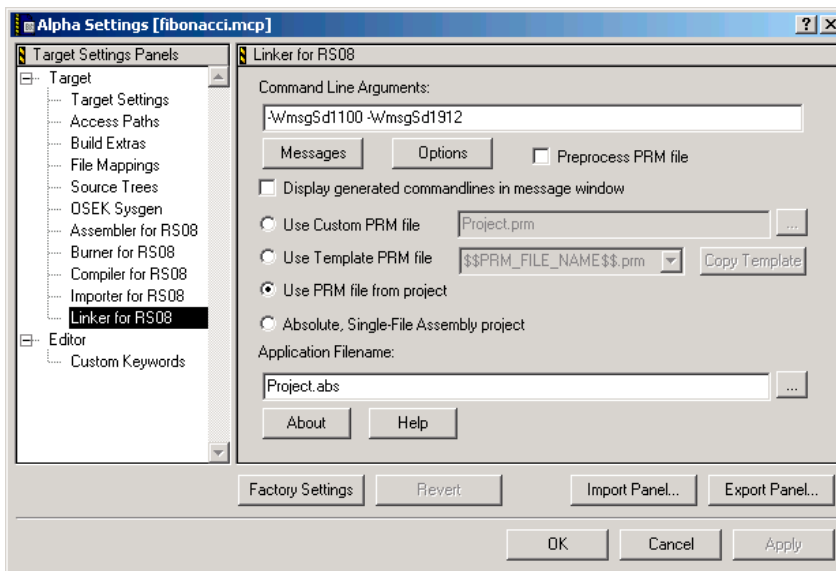


This causes the name of the Standard folder that contains the object files to be changed to Alpha. However, the names in the bin folder still are unchanged. You can change the name of the executable file to Alpha.abs by using another preference panel. From the *Edit* menu, select *Alpha Settings*. The *Alpha Settings* dialog box appears. Select *Target > Linker for RS08* in the *Target Settings Panels*. The *Linker for RS08* preference panel appears ([Figure 1.17](#)).

Introduction

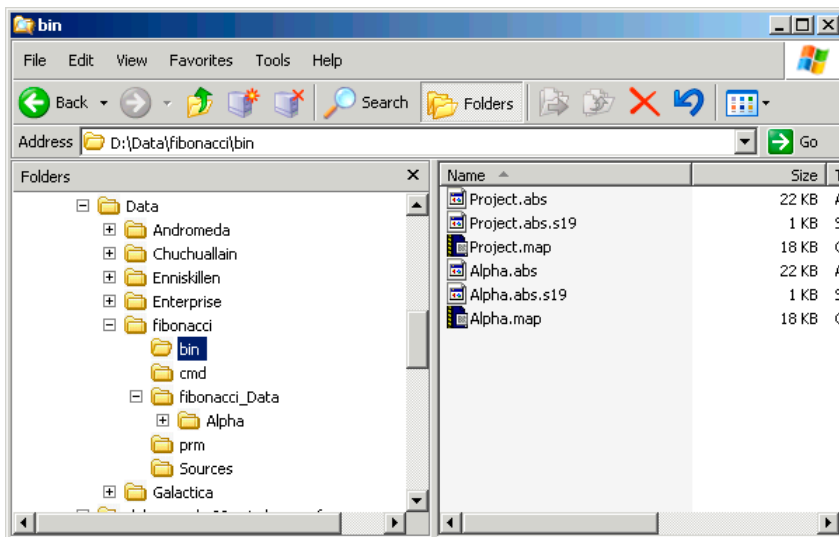
Using the CodeWarrior IDE to Manage a Project

Figure 1.17 Linker for RS08 Preference Panel



In the *Application Filename* text box, replace `Project.abs` with `Alpha.abs` and press *OK*. A dialog box appears stating that *Target 'Alpha' must be relinked*. Press *OK*. Press the *Make* icon on the Toolbar to rebuild the project. The contents of the `bin` folder change to reflect the new build target *Alpha* ([Figure 1.18](#)).

Figure 1.18 bin Folder after Changing Project to Alpha



Now, files with the Alpha.* filenames are generated. The previous Project.* files are not modified at all. However, they no longer are included in the project, so they may be safely deleted.

The Linker PRM file

The PRM file determines how the Linker allocates the RAM and ROM memory areas. The usual procedure is to use the default PRM file in the project window for any particular CPU derivative. However, it is possible to modify the PRM file if you want an alternative allocation.

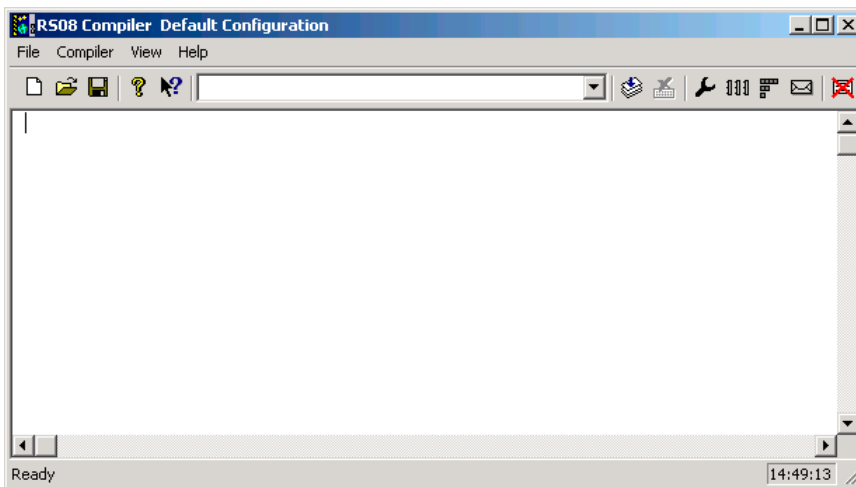
Compilation

You can use the RS08 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 *New Project Wizard*. Using the CodeWarrior software, you can create, configure, and manage a project much easier and quicker than using the Build Tools. However, you can use the Build Tools to 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 by opening the `crs08.exe` file in the `prog` folder in the RS08 CodeWarrior installation. The Compiler opens ([Figure 1.19](#)).

Figure 1.19 RS08 Compiler Opens



Read the Tips or 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, such as the listing of the most recent projects. 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`). Typical locations for a global configuration file are:

- `<CW installation directory>\prog\mcutools.ini`
- `C:\WINDOWS\mcutools.ini` - (used if no `mcutools.ini` file exists in above directory)

If you start a tool in the `C:\Program Files\Freescale\CodeWarrior for Microcontrollers V6.1\prog` directory, the initialization file in the same directory as the tool is used:

```
C:\Program Files\Freescale\CodeWarrior for  
Microcontrollers V6.1\prog\mcutools.ini.
```

If you start the tool from the CodeWarrior installation directory, the tool uses the initialization file in the Windows directory. For example,

```
C:\WINDOWS\mcutools.ini.
```

For information about entries in the global configuration file, see [Global Configuration File Entries](#) in the Appendices.

- Local

Any build tool can use this file for a particular project. For information about entries for the local configuration file, see [Local Configuration File Entries](#) in the Appendices.

Normally after opening the compiler, you load the configuration file for your project if it already has one. However, in this case you will create a new configuration file and save it so that when the project is reopened, it reuses the previously saved configuration state.

1. In Windows Explorer, create a new folder called `fibonacci_2`.
2. From the compiler *File* menu, select *New / Default Configuration*.

Now save this configuration in the newly created folder that will become the project directory.

3. From the *File* menu, select *Save Configuration* (or *Save Configuration As*).

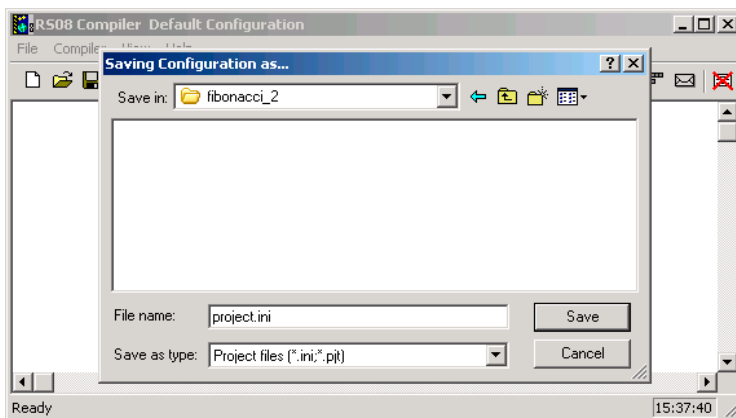
A *Saving Configuration as* dialog box appears.

Introduction

Compilation

4. Rename the file if you wish and navigate to the newly created folder ([Figure 1.20](#)).

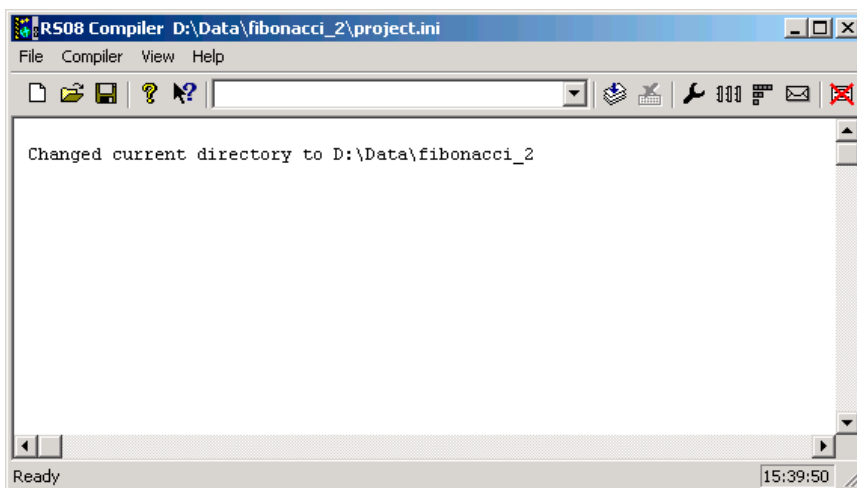
Figure 1.20 Loading Configuration Dialog Box



5. Press *Save*.

The current directory of the RS08 Compiler changes to your new project directory ([Figure 1.21](#)), and saves `project.ini` in the folder.

Figure 1.21 Compiler's Current Directory Switches to Your Project Directory



The project directory, examined using Windows Explorer, contains the `project.ini` configuration file that you just created. If you examine the contents of `project.ini`

configuration file, you notice that it now contains the `[CRS08_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 are placed into or deleted from this configuration file in the appropriate section for each Build Tool.

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

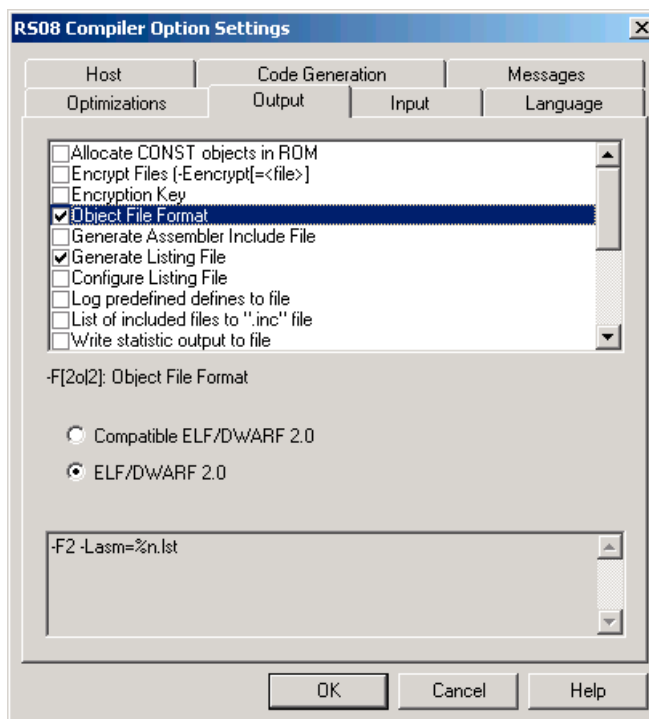
Now set the object file format that you intend to use (ELF/DWARF).

1. Select the *Compiler > Options > Options*.

The Compiler displays the *RS08 Compiler Option Settings* dialog box.

2. Select the *Output* tab ([Figure 1.22](#)).

Figure 1.22 Compiler Option Settings Dialog Box



3. In the *Output* panel, select the check boxes labeled *Generate Listing File* and *Object File Format*.
4. For the *Object File Format*, select the *ELF/DWARF 2.0* button.
5. Press *OK* to close the *RS08 Compiler Option Settings* dialog box.

Introduction

Compilation

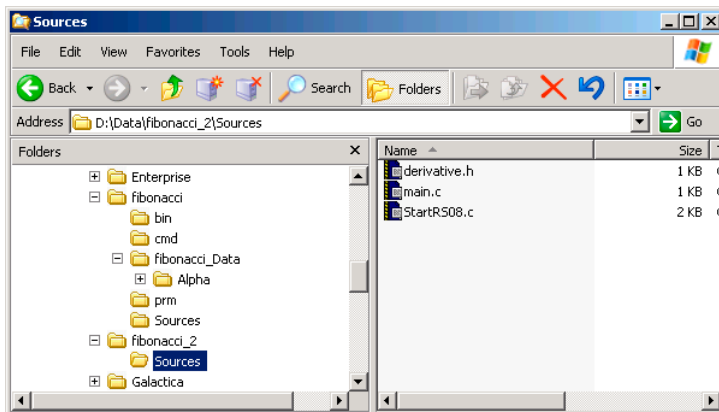
Save the changes to the configuration by:

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

Input Files

Now that the project's configuration is set, you can compile a C source-code file. You can create C source (**.c*) and include (**.inc*) files from scratch for this project, or copy and paste the Sources folder from a previous CodeWarrior project into the `fibonacci_2` project directory ([Figure 1.23](#)). In this case, copy the Sources folder from `fibonacci` into `fibonacci_2`.

Figure 1.23 Project Files



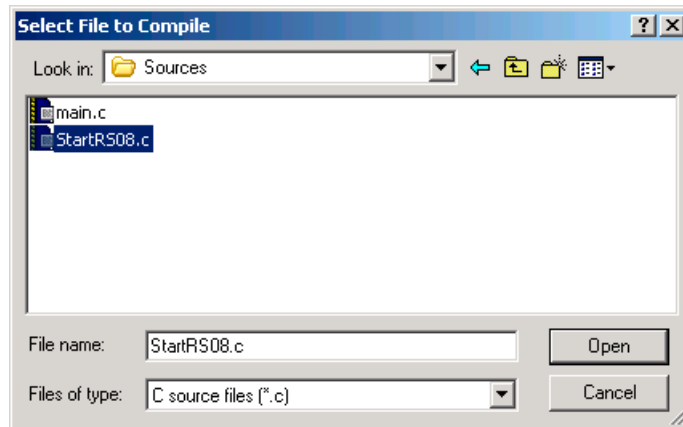
Now there are four files in the project:

- the `project.ini` configuration file in the project directory and
- in the Sources folder:
 - `derivative.h`
A collection of paged data-access runtime routines
 - `main.c`
The user's program plus derivative-specific and memory-model includes
 - `StartRS08.c`
The startup and initialization routines

Compiling the C Source-Code Files

Let's compile the `StartRS08.c` C source file. In the compiler, select *File > Compile*. The *Select File to Compile* dialog box appears ([Figure 1.24](#)).

Figure 1.24 Select File to Compile Dialog Box

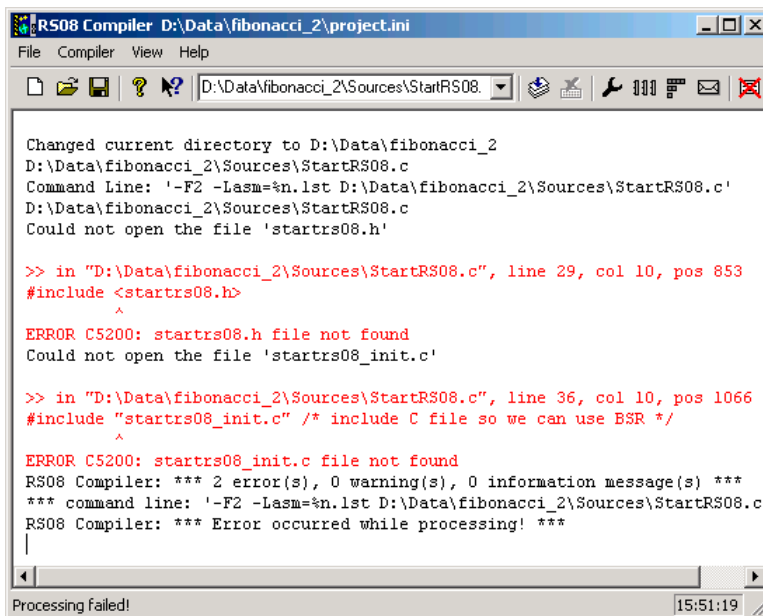


Browse to the *Sources* folder in the project directory and select the `StartRS08.c` file. Press *Open* and the `StartRS08.c` file starts compiling ([Figure 1.25](#)).

Introduction

Compilation

Figure 1.25 Results of compiling the StartRS08.c file



```

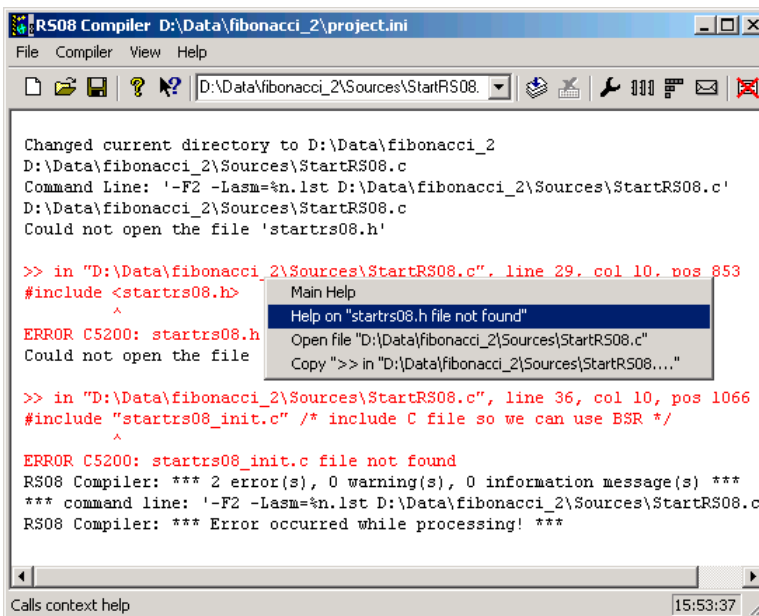
RS08 Compiler D:\Data\fibonacci_2\project.ini
File Compiler View Help
D:\Data\fibonacci_2\Sources\StartRS08.c
Changed current directory to D:\Data\fibonacci_2
D:\Data\fibonacci_2\Sources\StartRS08.c
Command Line: '-F2 -Lasm=%n.lst D:\Data\fibonacci_2\Sources\StartRS08.c'
D:\Data\fibonacci_2\Sources\StartRS08.c
Could not open the file 'starttrs08.h'

>> in "D:\Data\fibonacci_2\Sources\StartRS08.c", line 29, col 10, pos 853
#include <starttrs08.h>
      ^
ERROR C5200: starttrs08.h file not found
Could not open the file 'starttrs08_init.c'

>> in "D:\Data\fibonacci_2\Sources\StartRS08.c", line 36, col 10, pos 1066
#include "starttrs08_init.c" /* include C file so we can use BSR */
      ^
ERROR C5200: starttrs08_init.c file not found
RS08 Compiler: *** 2 error(s), 0 warning(s), 0 information message(s) ***
*** command line: '-F2 -Lasm=%n.lst D:\Data\fibonacci_2\Sources\StartRS08.c'
RS08 Compiler: *** Error occurred while processing! ***
|
Processing failed! 15:51:19
  
```

The project window provides positive or negative feedback information about the compilation process or generates error messages if compiling was unsuccessful. In this case two error messages are generated - two instances of the *C5200: 'FileName' file not found* message. If you right-click on the text about the error message, a context menu appears ([Figure 1.26](#)).

Figure 1.26 Context Menu



Select *Help on 'FileName' file not found* and help for the C5200 error message appears (Figure 1.27).

Figure 1.27 C5200 Error Message



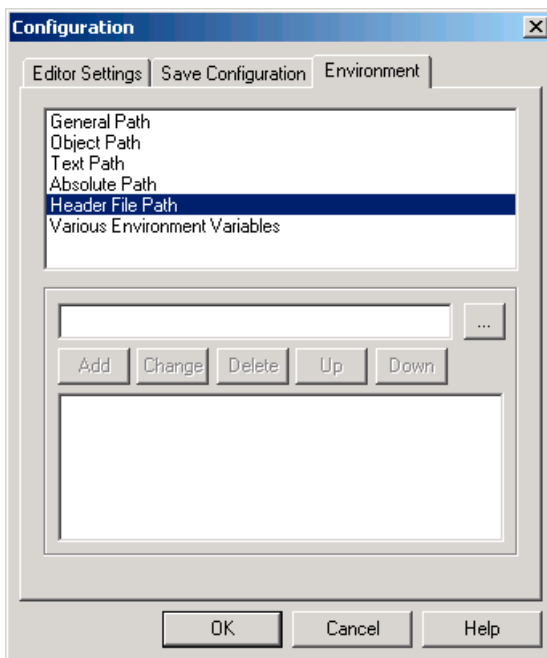
The *Tips* portion in the *Help for the C5200 error* tells you to specify the correct paths and names for the source files. Locate `Startrs08.h` in `<CodeWarrior installation folder>\lib\rs08c\include`. Locate `Startrs08_init.c` in `<CodeWarrior installation folder>\lib\rs08c\src`.

NOTE If you read the `Startrs08.c` file, you could have anticipated this because of an `#include` preprocessor directive for the header file. The other missing file was included by the header file.

The Compiler needs a configuration modification so that it can find these missing files.

1. Select *File > Configuration*.
The *Configuration* dialog box appears ([Figure 1.28](#)).

Figure 1.28 Browsing for the include Subfolder in the CodeWarrior lib Folder



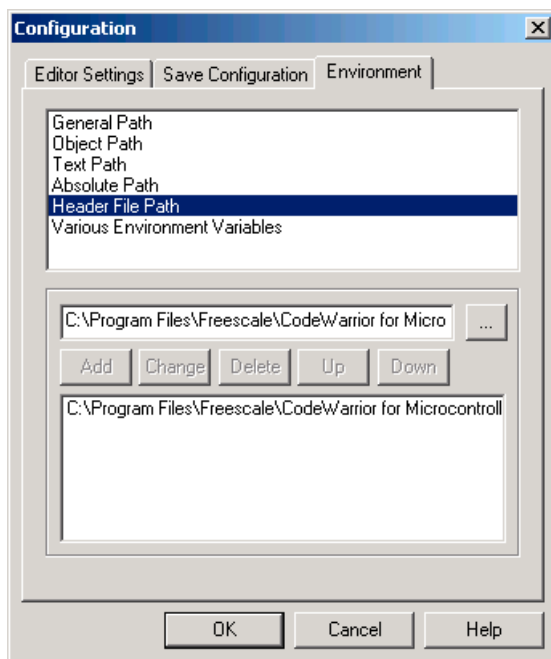
2. Select the *Environment* tab in the *Configuration* dialog box
3. Select *Header File Path*.
4. Press the “...” button and navigate in the *Browse for Folder* dialog box for the folder that contains the missing file: the `rs08c\include` subfolder in the CodeWarrior installation’s `lib` folder.
5. Press *OK* to close the *Browse for Folder* dialog box.

The *Configuration* dialog box is now active ([Figure 1.29](#)).

Introduction

Compilation

Figure 1.29 Adding a Header File Path



6. Press the *Add* button.

The path to the header file `C:\Program Files\Freescale\CodeWarrior for Microcontrollers V6.1\lib\rs08.c\include` now appears in the lower panel.

7. Press *OK*.

An asterisk now appears in the *Configuration Title* bar, so save the configuration modification.

8. Press the *Save* button or select *File > Save Configuration*.

If you do not save the configuration, the Compiler reverts to the last-saved configuration the next time the project is opened. After saving the asterisk disappears.

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

Repeat steps 1 through 8 to supply the path to `startrs08_init.c`, which is located in `C:\Program Files\Freescale\CodeWarrior for Microcontrollers V6.1\lib\rs08c\src`

Now that you have supplied the path to the missing files, you can try again to compile the `startrs08.c` file. Instead of compiling each file separately, compile both of them simultaneously.

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

Once again the compiler error messages indicate missing files, `hedef.h` and `derivative.h`. Locate these files in the installation or project folder, and add them to the configuration the same way you added `startrs08_init.c` and `startrs08.h`.

NOTE Some RS08-specific header files and source code files are located in the HC08 folders. If repeated errors occur, verify the file location using Windows, and add the correct file path to the configuration.

Select *File > Compile* and navigate to the Sources folder. Select both `*.c` files and press *Open*.

The Compiler indicates successful compilation of both C-source files and displays the Code Size for each. Also, the header files included by each C-source file are shown. The message `*** 0 error(s)`, indicates that the file compiled without errors. Do not forget to save the configuration one additional time.

The Compiler also generated object files in the Sources folder (for further processing by the Linker), and a output listing file in the project directory. The binary object file has the same name as the input module, but with the `*.o` extension instead. The assembly output file for each C-source file is similarly named.

NOTE The Compiler generates object-code files in the same location as the C-source files. If any C-source code file is in a CodeWarrior library folder (a subfolder inside `\lib`), we recommend that you configure the path for this C source file into somewhere other than this `lib` folder. The `OBJPATH` environment variable is used for this case. You use the *Object Path* option in the Configuration dialog box for this ([Figure 1.29](#)).

The haphazard running of this project was intentionally designed to fail in order to illustrate what occurs if the path to 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.

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.

Linking with the Linker

If you are using the standalone Linker (also known as the *SmartLinker*), use a PRM file for the Linker to allocate RAM and ROM memory areas:

1. Start your editor and create the project's linker parameter file. You can modify a *.prm file from another project and rename it as <target_name>.prm.
2. Store the PRM file in the project directory.
3. In the <target_name>.prm file, use the LINK section to add the name of the executable (*.abs) file, <target_name>.abs (use unique names for your *.abs files):

```
LINK Alpha.abs
```

4. Add the names of the object code files using the NAMES command:

```
NAMES StartRS08.c.o Main.c.o MC9RS08KA1.c.o END
```

NOTE You can also modify the start and end addresses for the ROM and RAM memory areas. The module's *.prm file is a PRM file from another CodeWarrior project. In the project window, double-click on a .prm file to display contents.

NOTE If you are adapting a PRM file from a CodeWarrior project, you need to add the LINK portion and the NAMES portion for object filenames that are to be linked.

NOTE Most of the entries in the PLACEMENT section are not used in this simple project. Furthermore, a number of extra entries were deleted from the actual PRM file used in another CodeWarrior project. It does not matter if all of these entries are used or not. They were left in order to show what entries are available for your future projects.

The commands in the linker parameter file are described in detail in the Linker section of the Build Tool Utilities manual.

1. Start the Linker.

The SmartLinker tool is located in the prog folder in the CodeWarrior installation:
prog\linker.exe

2. Press *Close* to close the *Tip of the Day* dialog box.
3. Load the project's configuration file.

Use the same <project>.ini that the Compiler used for its configuration - the project.ini file in the project directory:

4. Select *File > Load Configuration* and navigate to the project's configuration file.
5. Press *Open* to load the configuration file.

The project directory is now the current directory for the Linker. You can select *File > Save Configuration* to save the configuration if you choose. If you fail to save the configuration, the Linker reverts to its last-saved configuration when it is reopened. From the *File* menu in the SmartLinker, select *File > Link*.

6. Browse to locate the PRM file for your project.
7. Select the PRM file.
8. Press *Open*.

The Smart Linker links the object-code files in the NAMES section to produce the executable * .abs file as specified in the LINK portion of the Linker PRM file.

The messages in the linker's project window indicate:

- The current directory for the Linker is the project directory,
D:\Data\fibonacci_2
- The `project.prm` file was used to name the executable file, specify which object files were linked, and allocate the RAM and ROM memory areas for the relocatable sections.
- There were three object-code files, `Starttrs08.c.o`, `main.c.o`, and `MC9RS08KA1.C.o`.
- The output format was DWARF 2.0.
- The Code Size was 137 bytes.
- A Linker Map file was generated: `<project>.map`.
- No errors or warnings occurred and no information messages were issued.

Using the Simulator/Debugger

Use the Simulator/Debugger Build Tool, `hiwave.exe`, located in the `prog` folder in the CodeWarrior installation to simulate the sample program in the `main.c` source-code file. The Simulator Build Tool can be operated in this manner:

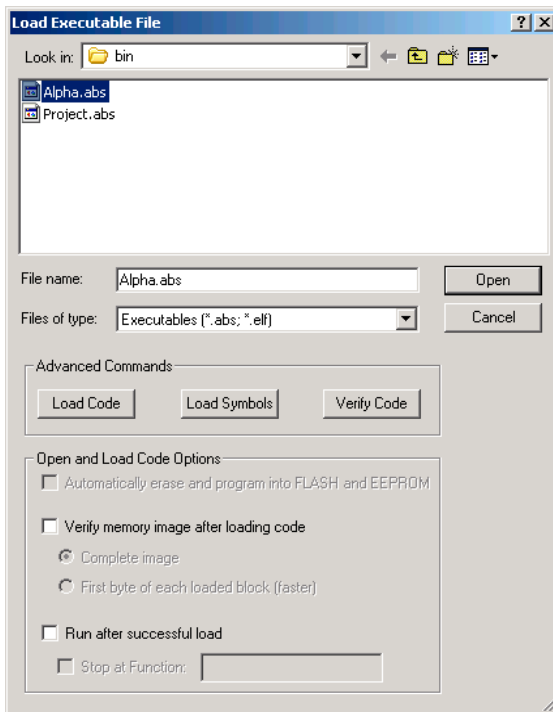
- Start the Simulator.
- Load the absolute executable file:
 - *File > Load Application* and browse to the appropriate * .abs file or
 - Select the given path to the executable file, if it is appropriate as in this case ([Figure 1.30](#)):

```
D:\Data\fibonacci\alpha.abs
```

Introduction

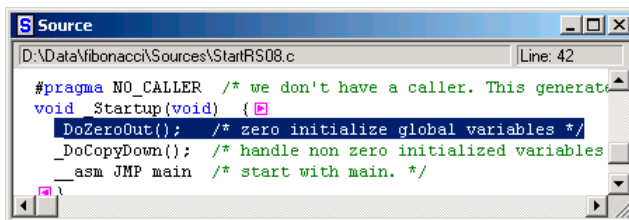
Compilation

Figure 1.30 RS08 Simulator: Select the Executable File



- Assembly-Step ([Figure 1.31](#)) through the program source code.

Figure 1.31 Simulator Startup



You can simulate this particular C program through its program, to gain an insight as to what the `startrs08.c` routines are before it turns the program over to the routines in `main.c`.

Application Programs (Build Tools)

You will 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:\Program Files\Freescale\` directory, all the Build Tools are located in the `C:\Program Files\Freescale\prog` directory with the exception of `IDE.exe` which is found in the `bin` subfolder of the CodeWarrior installation folder.

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

- `IDE.exe` - CodeWarrior IDE
- `crs08.exe` - Freescale RS08 Compiler
- `ahc08.exe` - Freescale HC08/RS08 Assembler
- `libmaker.exe` - Librarian Tool to build libraries
- `linker.exe` - Link Tool to build applications (absolute files). The Linker is also referred to as the *Smart Linker*.
- `decoder.exe` - Decoder Tool to generate assembly listings. This is another name for a *Disassembler*.
- `maker.exe` - Make Tool to rebuild automatically
- `burner.exe` - Batch and interactive Burner
- `hiwave.exe` - Multi-Purpose Simulation or Debugging Environment
- `piiper.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 1.1](#)).

Listing 1.1 Example of a startup command-line option

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

Introduction

Highlights

There are other options that launch a build 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 1.2](#)) 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.

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 1.2 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

All required plug-ins are installed together with the CodeWarrior IDE. The CodeWarrior IDE is installed in the `bin` directory (usually `C:\Freescale\CodeWarrior for Microcontrollers V6.1\bin`). The plug-ins are installed in the `bin\plugins` directory.

Combined or Separated Installations

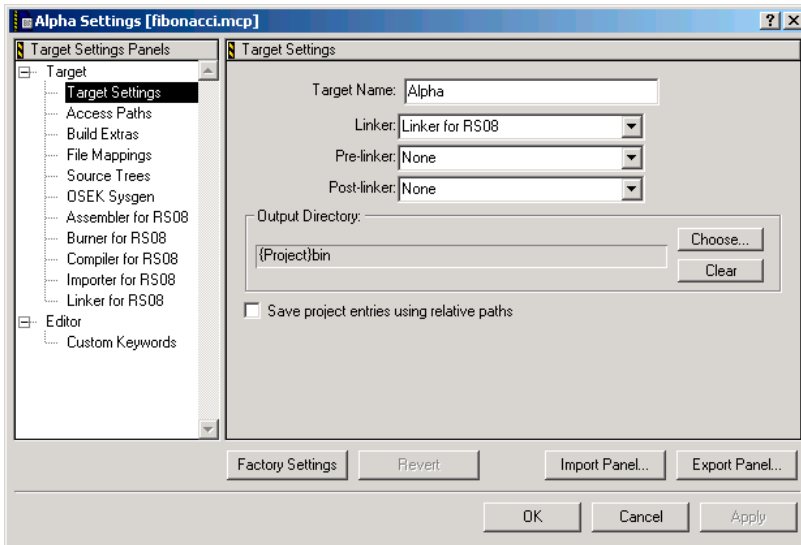
The installation script enables you to install several CPUs in one single installation path. This saves disk space and enables switching from one processor family to another without leaving the IDE.

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:\Freescale\CodeWarrior for Microcontrollers V6.1\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 1.32](#)).

Figure 1.32 Target Settings Preference Panel



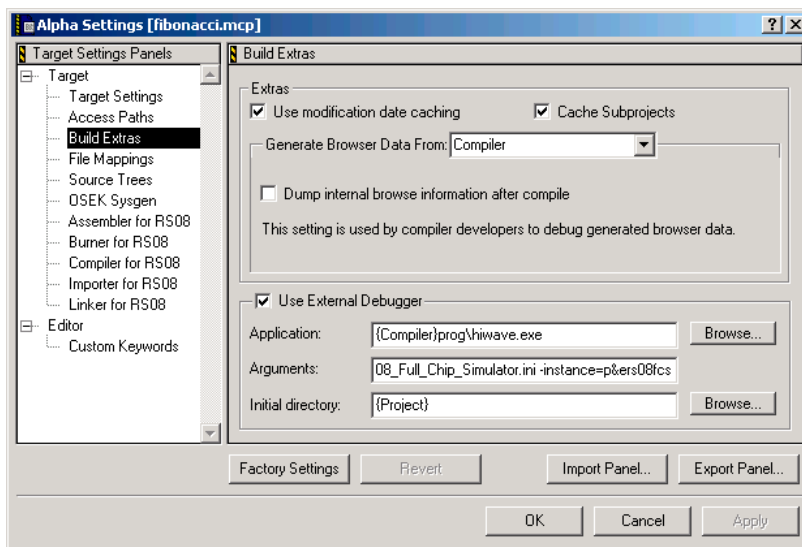
Depending on the CPU targets installed, you can choose from various linkers available in the linker drop box.

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

Use the Build Extras Preference Panel ([Figure 1.33](#)) to get the compiler to generate browser information.

Figure 1.33 Build Extras Preference Panel



Enable the *Use External Debugger* check box to use the external simulator or debugger. Define the path to the debugger, which is either absolute (for example, C:\Program Files\Freescale\CodeWarrior for Microcontrollers V6.1\prog\hiwave.exe), or installation-relative (for example, {Compiler}prog\hiwave.exe).

Introduction

CodeWarrior Integration

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

Table 1.1 Additional % macros for the external debugger

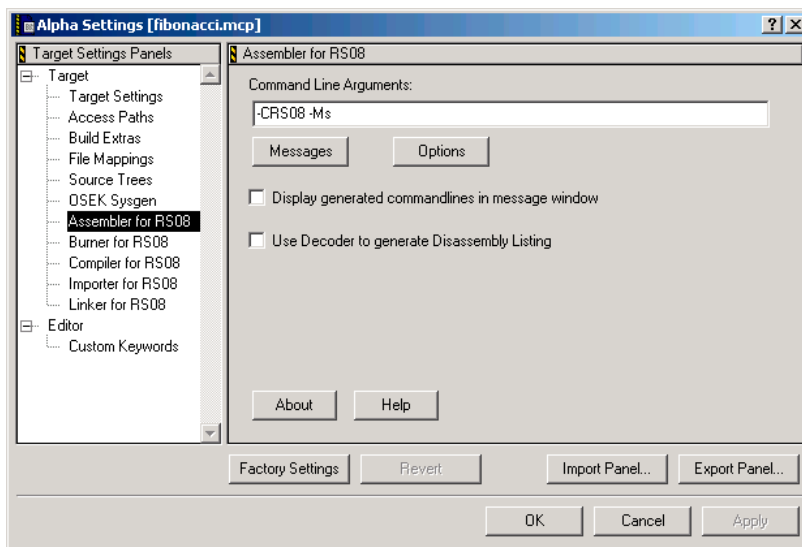
%sourceFilePath	%projectSelectedFiles
%sourceFileDir	%targetFilePath
%sourceFileName	%targetFileDir
%sourceLineNumber	%targetFileName
%sourceSelection	%currentTargetName
%sourceSelUpdate	%symFilePath
%projectFilePath	%symFileDi
%projectFileDir	%symFileName
%projectFileName	

Assembler for RS08 Preference Panel

The Assembler for RS08 preference panel ([Figure 1.34](#)) 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.
 - 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.

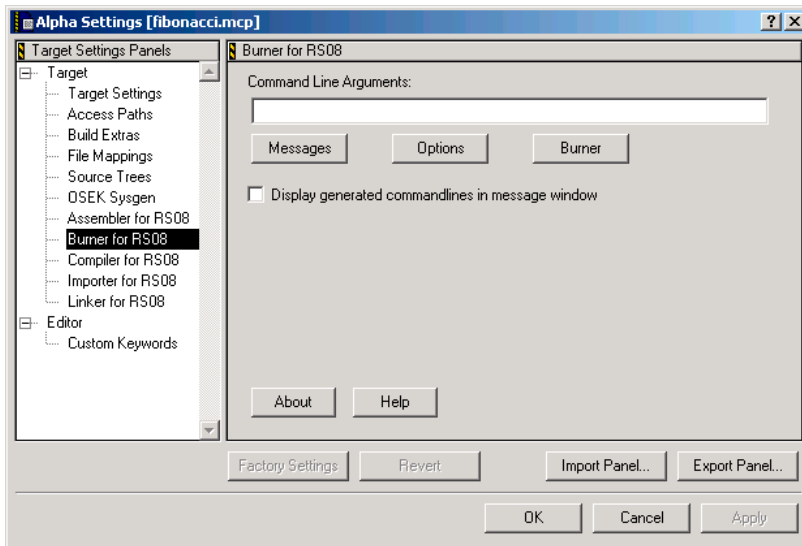
Figure 1.34 Assembler for RS08 Preference Panel



Burner 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 1.35](#)).

Figure 1.35 Burner for RS08 Preference Panel



The Burner for RS08 preference panel contains the following:

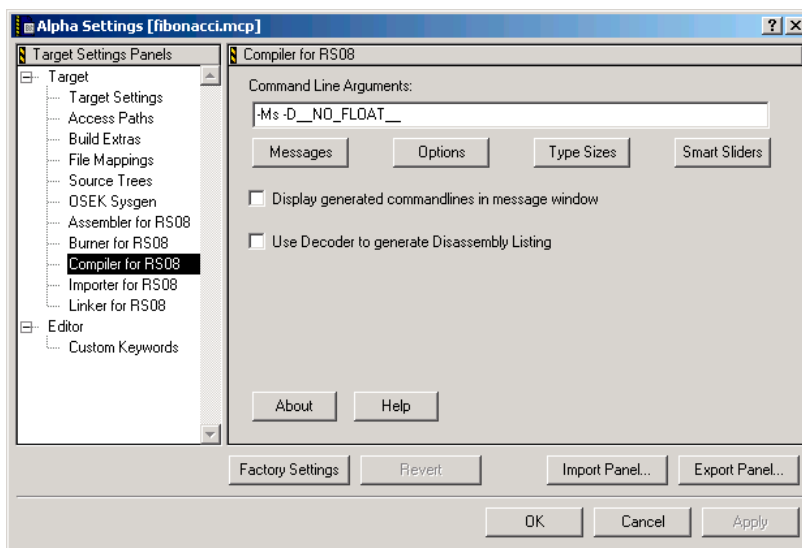
- *Command Line Arguments*: The actual command line options are displayed. You can add, delete, or modify the options manually, or use the Messages, Options, and Burner buttons.
 - *Messages*: Opens the Messages dialog box
 - *Options*: Opens the Options dialog box
 - *Burner*: Opens the Burner dialog 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 RS08 Preference Panel

The plug-in Compiler Preference Panel ([Figure 1.36](#)) contains the following:

- *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.

Figure 1.36 Compiler for RS08 Preference Panel

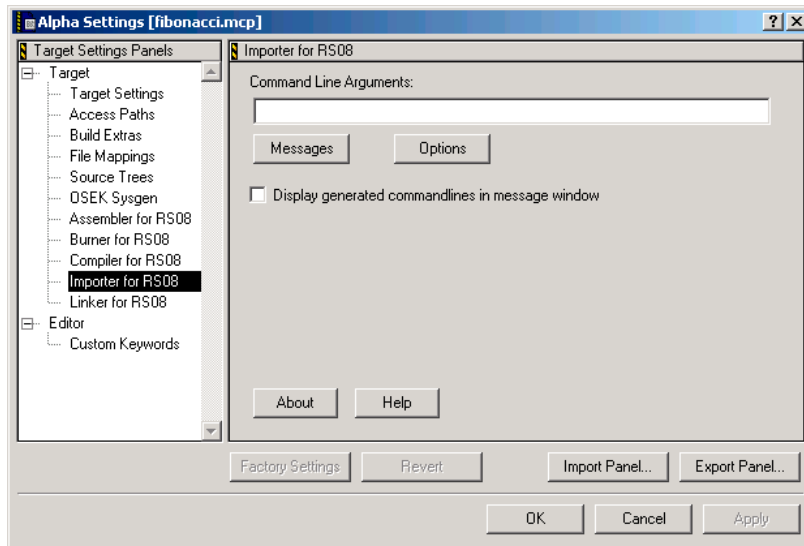


Importer for RS08 Preference Panel

The plug-in Importer Preference Panel ([Figure 1.37](#)) 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.

Figure 1.37 Importer Preference Panel

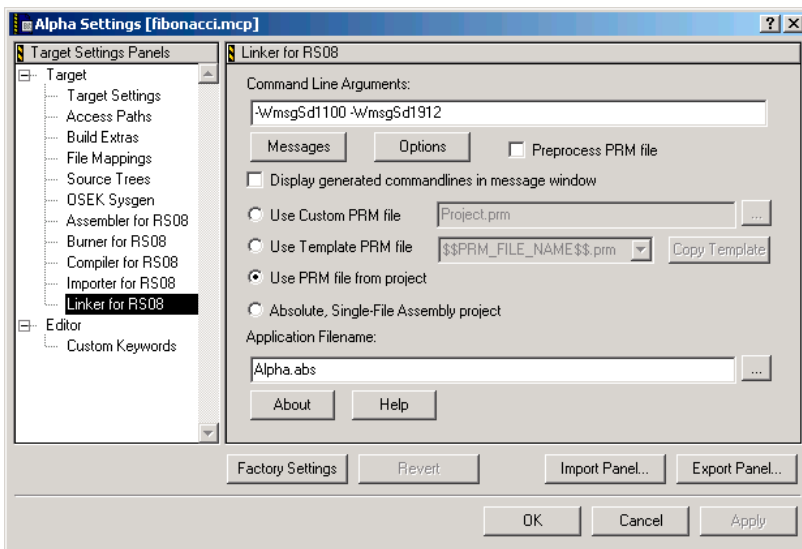


Linker for RS08 Preference Panel

This preference panel ([Figure 1.38](#)) displays in the *Target Settings Panel* if the Linker is selected. The plug-in 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
- *Preprocess PRM file*: When checked, the preprocessor of the ANSI-C compiler is used to preprocess the PRM file prior to the linking step. In the PRM file, all ANSI-C preprocessor conditions like conditional inclusion (`#if`) are available. The same preprocessor macros as in ANSI-C code can be used (e.g., `#ifdef __SMALL__`).
- *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\). 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.

Figure 1.38 Linker Preference Panel



CodeWarrior Tips and Tricks

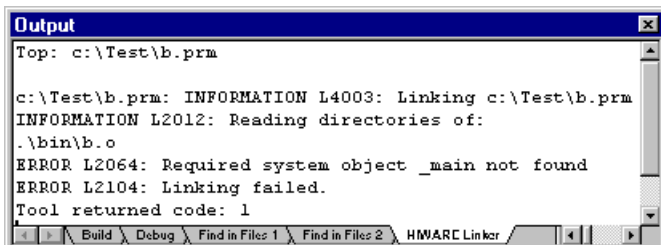
If the Simulator or Debugger cannot be launched, check the settings in the *Build Extras Preference Panel*.

If the data folder of the project is deleted, then some project-related settings may also have been deleted.

If a file cannot be added to the project, its file extension may be absent from the *File Mappings Preference Panel*. Add this file's extension to the listing in the *File Mappings Preference Panel* to 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*.

Figure 1.39 Compiler Log Display



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

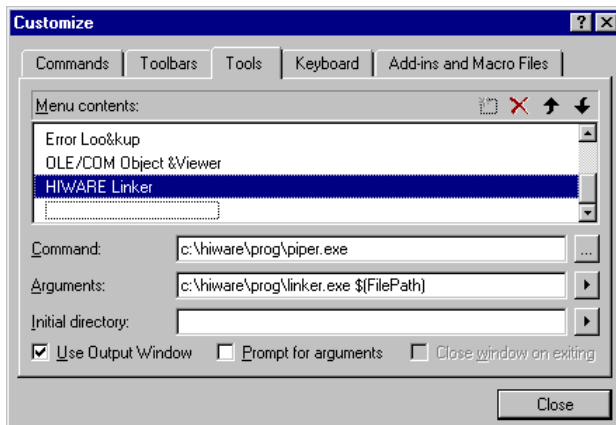
Use the following procedure to integrate the Tools into the Microsoft Visual Studio (Visual C++).

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*.

Proceed as above for all other tools (for example, compiler, decoder).

Close the *Customize* dialog box ([Figure 1.40](#)).

Figure 1.40 Customize Dialog Box

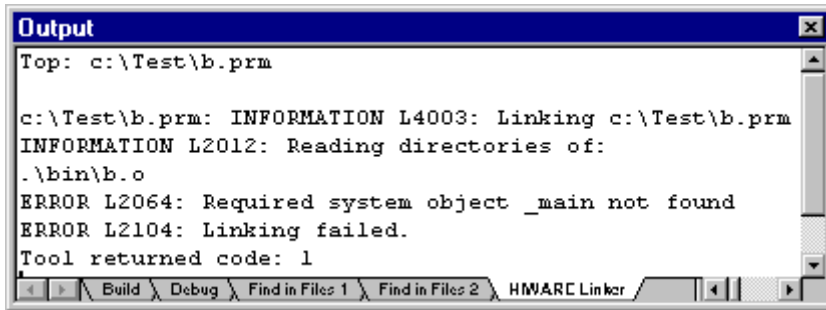


Introduction

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

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 1.41). Double click on the output entries to jump to the right message position (message feedback).

Figure 1.41 Visual Output Window



Use the following procedure to integrate the Toolbar in Microsoft Visual Studio (Visual C++).

1. Start Visual Studio.
 - Make sure that all tools are integrated as *Additional Tools*.
2. Select the menu *Tools > Customize*.
3. Select the *Toolbars* Tab.
4. Select *New* and enter a name (for example, *CodeWarrior Build Tools*). A new empty toolbar named *CodeWarrior Build Tools* appears on your screen.
5. Select the *Commands* Tab.
6. In the *Category* drop down box, select *Tools*.
 - 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.)
7. Drag the selected tool image to the *CodeWarrior Build Tools* toolbar.
 - All default tool images look the same, making it difficult to know which tool has been launched. Associate a name with them.
 - a. Right-click on a tool in the *CodeWarrior Build Tools* to open the context menu of the button.
 - b. Select *Button Appearance* in the context menu.
 - c. Select *Image and Text*.

- d. Enter the tool name to associate with the image in *Button text*: (for example, *Linker*).
8. Repeat the above for all the desired tools to appear in the toolbar.
9. Close the *Customize* dialog box after all the Build Tools are entered into the Toolbar.

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.
- language extensions that are specially designed for more efficient embedded systems programming.

Object-File Formats

The Compiler supports only the ELF/DWARF 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 Do not mix object-file formats. Both HIWARE and the ELF/DWARF object files use the same filename extensions. HIWARE object-file format is not supported on the RS08.

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.

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

Introduction

Object-File Formats

Tools

The CodeWarrior Development Studio contains the following Tools, among others:

Compiler

The same Compiler executable supports both object-file formats. Use the [-F \(-F2, -F2o\): Object-File Format](#) compiler option to switch the object-file format.

Note that the RS08 Compiler backend supports only the ELF/DWARF object-file format, not the HIWARE object-file format. Some compiler backends support one or both.

Decoder

Use the executable `decoder.exe` for the ELF/DWARF object-file format.

Linker

Use the executable `linker.exe` for the ELF/DWARF object-file format.

Simulator or Debugger

The Simulator or Debugger supports the ELF/DWARF object-file format.

Mixing Object-File Formats

Mixing HIWARE and ELF object files is not possible. HIWARE object file formats are not supported on the RS08. 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, the CodeWarrior IDE, CodeWright, MS Visual Studio, or 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](#)
- [Specifying the Input File](#)

Launching the Compiler

Start the compiler using:

- The Windows Explorer
- An Icon on the desktop
- 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\crs08.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\crs08.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\crs08.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 2000®.

Windows XP, or Windows Vista™ operating systems, or use the `/wait` options (see Windows Help `help start`). Using `start /wait` ([Listing 2.4](#)) you can write perfect batch files.

Listing 2.4 Start a compilation process and wait for termination

```
C:\> start /wait C:\Freescale\prog\crs08.exe -F2 a.c d.c
```

Tip of the Day

When you start the application, a standard *Tip of the Day* ([Figure 2.1](#)) window opens 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.

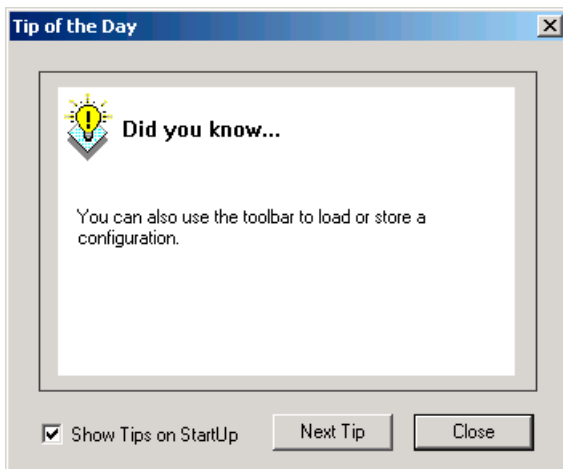
To enable automatic display from the standard *Tip of the Day* window when the application is started, select the entry *Help > Tip of the Day*. The *Tip of the Day* window opens. Check the box *Show Tips on Start Up*.

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

Graphical User Interface

Main Window

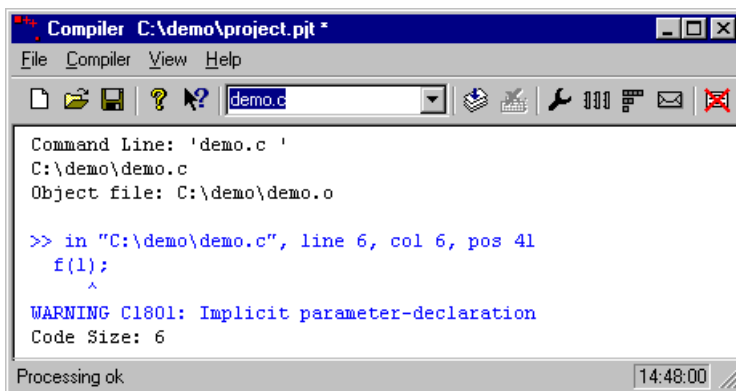
Figure 2.1 Tip of the Day Dialog



Main Window

This Main Window ([Figure 2.2](#)) is only visible on the screen 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, *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:

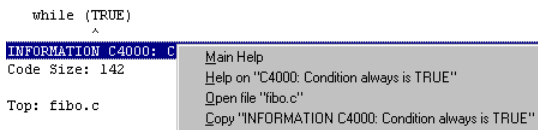
- A filename including a position inside of a file
- A message number

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 appears.
- Press *Shift-F1* and then click on the message text.
If the message text does not have a message number, the main help appears.
- 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 Online Help Dialog



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 FI*), 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



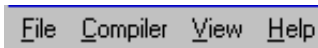
Menu Bar

[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

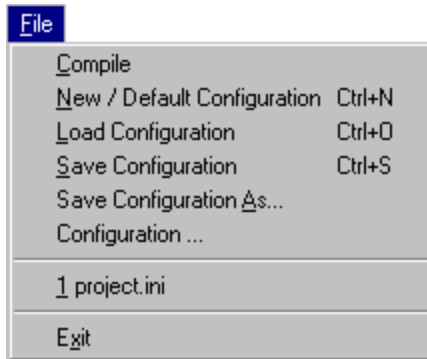


File Menu

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.

[Table 2.2](#) describes all the commands that are available from the File Menu:

Table 2.2 File Menu Commands

Menu entry	Description
Compile	Opens a standard Open File box. The configuration data stored in the selected file is loaded and used by a future session.
New / Default Configuration	Resets the application option settings to the default value. The application options which are activated per default are specified in section <i>Command Line Options</i> in this document
Load Configuration	Opens a standard Open File box. The configuration data stored in the selected file is loaded and used by a future session.
Save Configuration	Saves the current settings.
Save Configuration As	Opens a standard Save As box. The current settings are saved in a configuration file which has the specified name. See Local Configuration File (usually project.ini) .
Configuration	Opens the <i>Configuration</i> dialog box to specify the editor used for error feedback and which parts to save with a configuration.

Graphical User Interface

Menu Bar

Table 2.2 File Menu Commands (continued)

Menu entry	Description
1... project.ini 2...	Recent project list. Access this list to reopen a recent project.
Exit	Closes the application.

Editor Settings dialog box

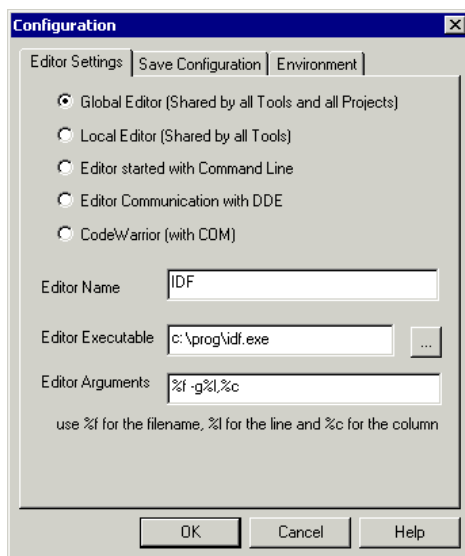
The Editor Settings dialog box has a main selection entry. Depending on the main type of editor selected, the content below changes.

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 global initialization file `mcutools.ini` in the [Editor] section. Some [Modifiers](#) are specified in the editor command line.

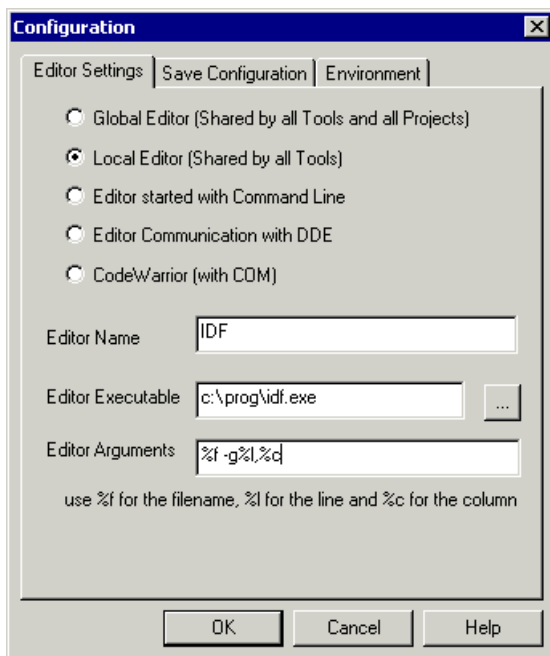
Figure 2.9 Global Editor configuration



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

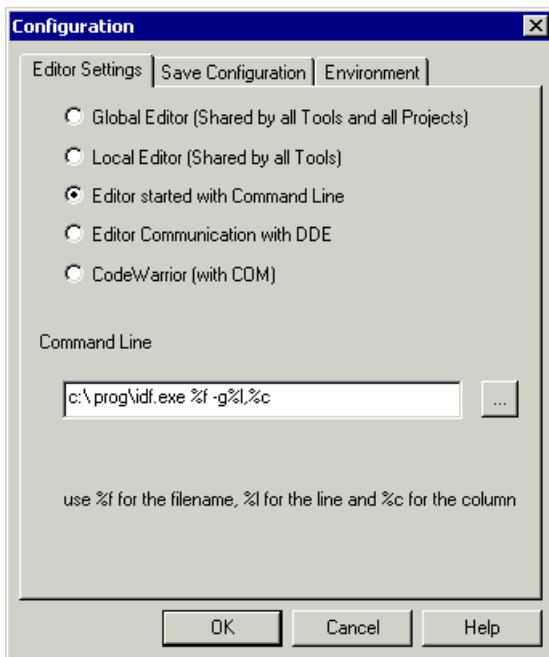
When this editor type ([Figure 2.11](#)) is selected, a separate editor is associated with the application for error feedback. The configured editor is not used for error feedback.

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:

For CodeWright V6.0 version, use (with an adapted path to the cw32.exe file):

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

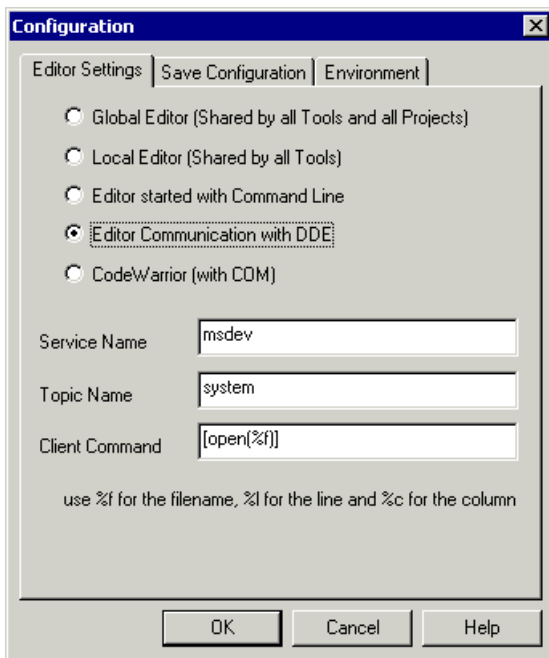
For the WinEdit 32-bit version, use (with an adapted path to the winedit.exe file):

```
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 Developer 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](#).

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 or www.ultraedit.com, email idm@idmcomp.com. For UltraEdit, use the following settings ([Listing 2.6](#)).

Graphical User Interface

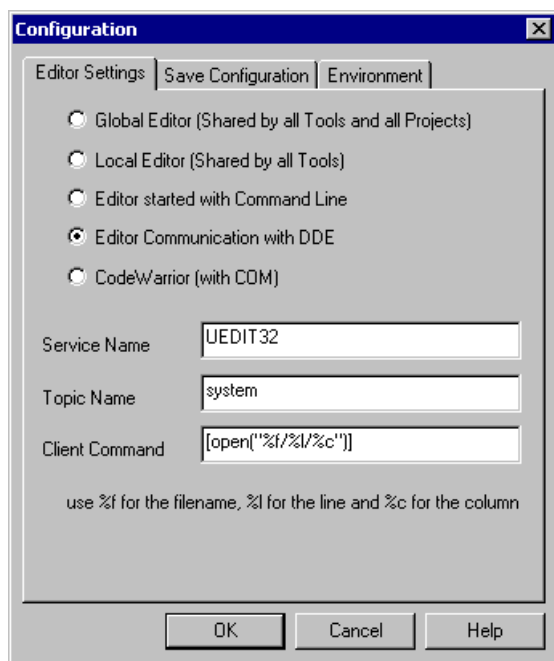
Menu Bar

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) must be started or the DDE communication fails.

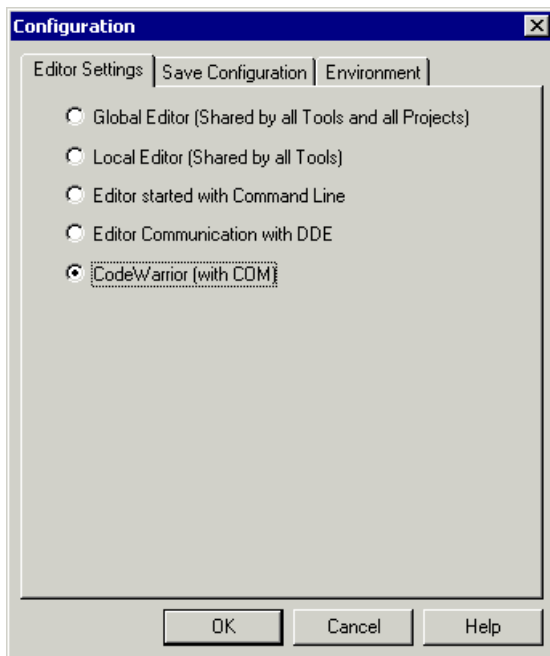
Figure 2.13 Editor Started with DDE (UltraEdit-32)



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\WINEEDIT\Winedit.EXE %f
Check the editor manual to define which command line to use to start the editor.

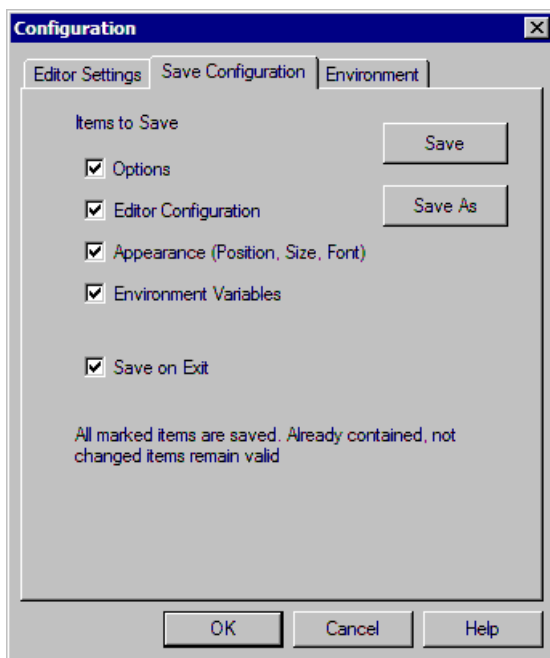
Save Configuration dialog box

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 are 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

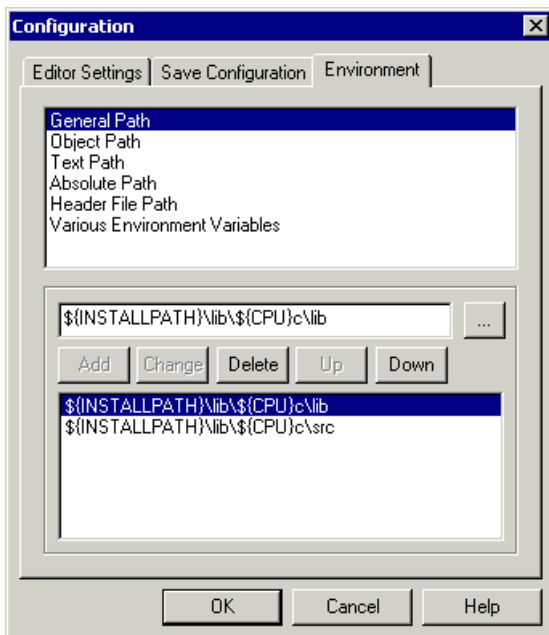
This Environment Configuration dialog box ([Figure 2.16](#)) is used to configure the environment. The content of the dialog box is read from the actual project file out of the section [Environment Variables].

The following environment variables are available ([Listing 2.1](#)):

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 2.3](#)):

Table 2.3 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.

Figure 2.17 Compiler Menu

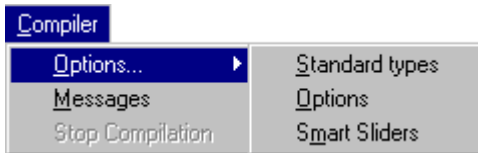


Table 2.4 Compiler Menu options

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 .)
Advanced	Allows you to define the options which must be activated when processing an input file. (See Option Settings dialog box .)
Smart Slider	Allows you to define the optimization level you want to reach when processing the input file. (See Compiler Smart Control dialog box .)
Messages	Opens a dialog box, where the different error, warning, or information messages are mapped to another message class. (See Message Settings dialog box .)
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 2.5](#) lists the View Menu options.

Figure 2.18 View Menu

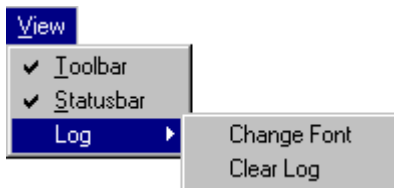


Table 2.5 View Menu options

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 2.6](#) defines the Help Menu options:

Figure 2.19 Help Menu

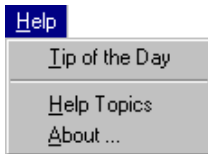


Table 2.6 Help Menu Options

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 incorrect to specify all types to a size of 32 bits. There is no type for 8 bits and 16 bits available for the Compiler.

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)
```

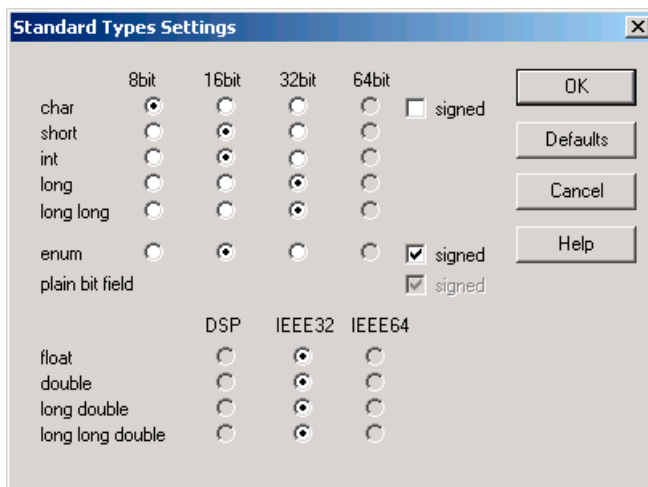
```
sizeof(double) <= sizeof(long double)
```

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 2.7](#) lists the Option Settings dialog box selections.

Figure 2.21 Option Settings dialog box

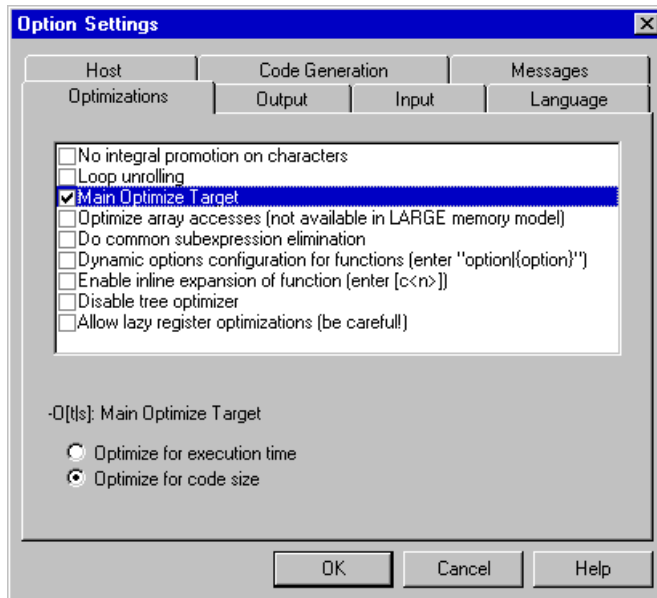


Table 2.7 Option Settings dialog box selections

Group	Description
Optimizations	Lists optimization options.
Output	Lists options related to the output files generation (which kind of file to generate).
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 (such as memory models or float format).
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 2.8](#).

Figure 2.22 Compiler Smart Control dialog box

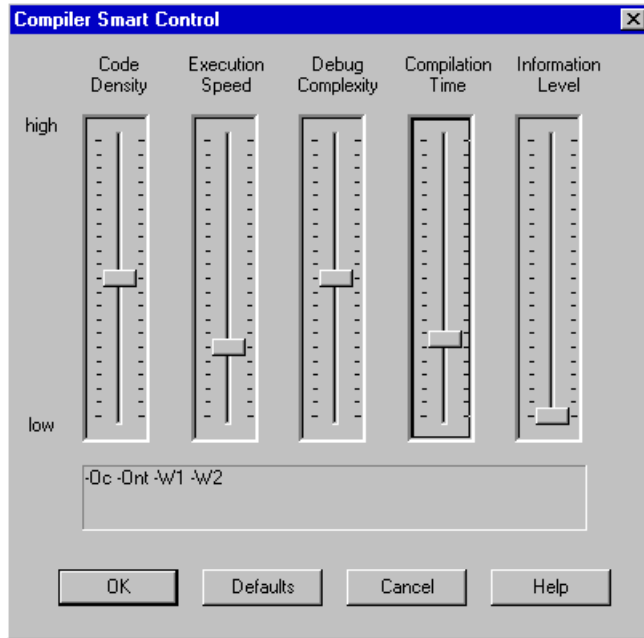


Table 2.8 Compiler Smart Control dialog box controls

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 2.9](#) lists and describes the buttons available in this dialog box.

Figure 2.23 Message Settings dialog box

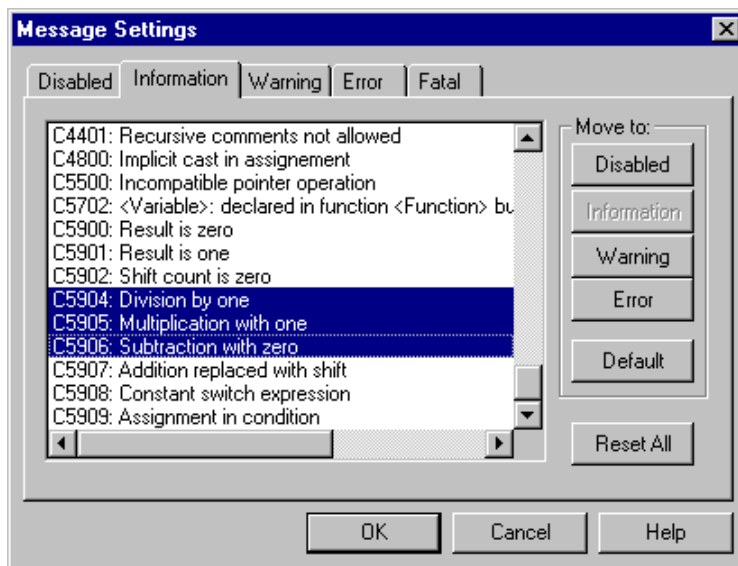


Table 2.9 Message Settings dialog box buttons

Button	Description
Move to: Disabled	The selected messages are disabled. The message will not occur any longer.
Move to: Information	The selected messages are changed to information messages.
Move to: Warning	The selected messages are changed to warning messages.
Move to: Error	The selected messages are changed to error messages.
Move to: Default	The selected messages are changed to their default message type.
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 2.10](#) lists the definitions for the message groups.

Table 2.10 Message Group Definitions

Message group	Description
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.
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:

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).

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 are 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 is 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.

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.

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.



Graphical User Interface

Specifying the Input File

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:

- 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.
- 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.
- 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>

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 environment variable [DEFAULTDIR: Default Current Directory](#).

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}

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

Environment

Global Initialization File (mcutools.ini)

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 are 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](#)).

Listing 3.4 EBNF path syntax

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

Most environment variables contain path lists directing where to look for files ([Listing 3.5](#)).

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.

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
```

This results 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.

COMPOPTIONS: Default Compiler Options

Tools

Compiler

Synonym

HICOMPOPTIONS

Syntax

COMPOPTIONS={<option>}

Arguments

<option>: Compiler command-line option

Environment

Environment Variable Details

Default

None

Description

If this environment variable is set, the Compiler appends its contents to its command line each time a file is compiled. Use this variable to specify global options for every compilation. 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](#)

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).

Environment

Environment Variable Details

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 a system level environment variable (global environment variable). It cannot be specified in a default environment file (`default.env`).

Example

```
ENVIRONMENT=\Freescale\prog\global.env
```

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

Environment

Environment Variable Details

`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.

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
```

Environment

Environment Variable Details

See also

Environment variables:

- [COPYRIGHT: Copyright entry in object file](#)
- [USERNAME: User Name in Object File](#)

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.

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**Environment variables:**

- [GENPATH: #include “File” Path](#)
 - [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 issues 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
```

Environment

Environment Variable Details

See also

[Output Files](#)

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](#)
- [Output Files](#)
- [File Processing](#)

Input Files

The following input files are described:

- [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>
```

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:

- [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](#)).

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 CodeWright'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 file named `err.txt` is generated in the current directory.

Batch Mode (Compiler Window not Open)

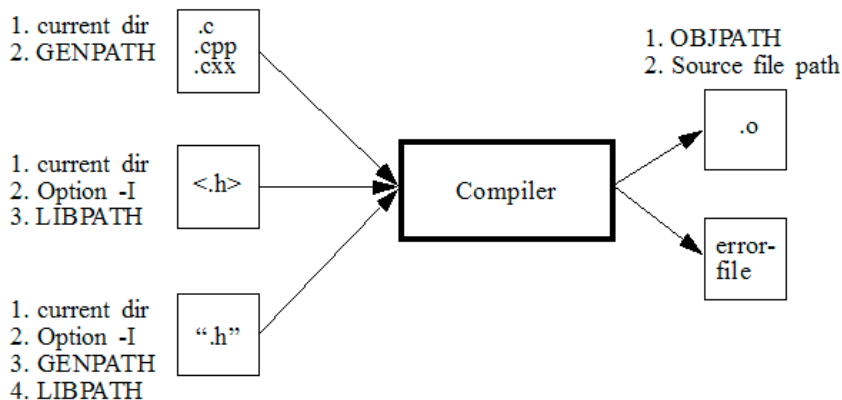
If `ERRORFILE` is set, the Compiler creates a message file named as specified in this environment variable.

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`.

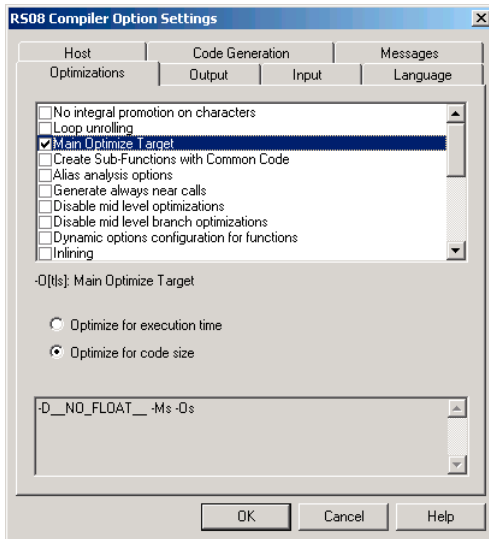
NOTE It is not possible to coalesce options in different groups, e.g., `-Cc -Li` cannot be abbreviated by the terms `-Cci` or `-Ccli`.

Another way to set the compiler options is to use the RS08 Compiler Option Settings dialog box ([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.

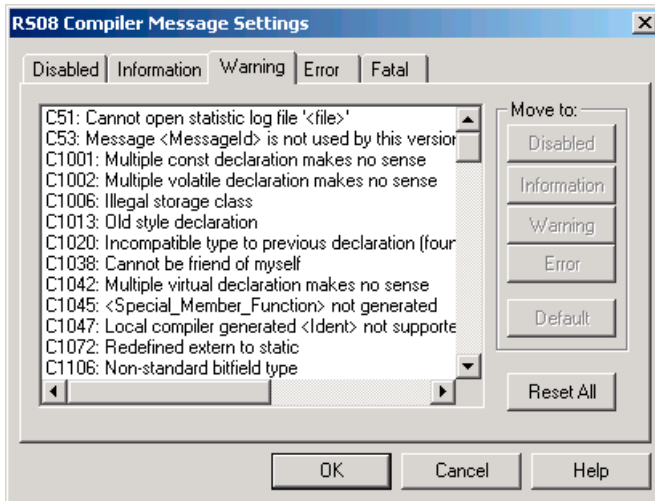
Compiler Options

Figure 5.1 Option Settings dialog box



The RS08 Compiler Message Settings dialog box, shown in [Figure 5.2](#), may also be used to move messages (`-Wmsg` options).

Figure 5.2 RS08 Compiler Message Settings dialog box



Option Recommendation

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

- [-Wpd: Error for Implicit Parameter Declaration](#)

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

Group	Description
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 output file generation options (types of file generated)
INPUT	Lists options related to the input file
CODE GENERATION	Lists options related to code generation (memory models, float format, etc.)
TARGET	Lists options related to the target processor
MESSAGES	Lists options controlling error message generation
VARIOUS	Lists various options
STARTUP	Options specified only 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

Scope	Description
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: "-OdocF=" "<option>".
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

Topic	Description
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

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

Examples

For the following examples, the actual base filename for the modifiers is:

```
C:\Freescale\my_demo\TheWholeThing.myExt.
```

`%p` gives the path only with a file separator:

```
C:\Freescale\my_demo\
```

`%N` results in the filename in 8.3 format (that is, the name with only eight characters):

```
TheWhole
```

`%n` returns just the filename without extension:

```
TheWholeThing
```

`%E` gives the extension in 8.3 format (that is, the extension with only three characters)

```
myE
```

`%e` is used for the whole extension:

Compiler Options

Compiler Option Details

myExt

%f gives the path plus the filename:

C:\Freescale\my demo\TheWholeThing

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

"C:\Freescale\my demo\TheWholeThing"

where %'%f%' results in: (using single quotes)

'C:\Freescale\my demo\TheWholeThing'

%(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

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

myfile.txt

A %% may be used to print a percent sign. Using %e%% results in:

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 (eight characters).

Listing 5.1 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 the included file**Default**

None

DefinesNone

Compiler Options

Compiler Option Details

Pragmas

None

Description

Includes the specified file at the beginning of the compilation unit. It has the same effect as if written at the beginning of the compilation unit using double quotes ("`...`"):

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

Example

See [Listing 5.2](#) for the `-AddIncl` compiler option that includes the above header file.

Listing 5.2 -AddIncl example

```
-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.

-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

`-BfaBLS`

Compiler Options

Compiler Option Details

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 5.3](#) uses the default condition and uses the three least significant bits.

Listing 5.3 Example struct used for the next listing

```
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 5.4](#).

Listing 5.4 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

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

Compiler Options

Compiler Option Details

to reach this. This option enables you to affect the maximum number of gap bits allowed.

Example

In the example in [Listing 5.5](#), it is assumed that you have specified a 3-bit maximum gap, i.e., `-BfaGapLimitBits3`.

Listing 5.5 Bitfield allocation

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

The compiler allocates struct B with three bytes. First, the compiler allocates the seven bits of a. Then the compiler tries to allocate the five bits of b, but this would cross a byte boundary. Because the gap of one bit is smaller than the specified gap of three bits, b is allocated in the next byte. Then the allocation starts for c. After the allocation of b there are three bits left. Because the gap is three bits, c is allocated in the next byte. If the maximum gap size were specified to zero, all 16 bits of B would be allocated in two bytes.

[Listing 5.6](#) specifies a maximum size of two bits for a gap.

Listing 5.6 Example where the maximum number of gap bits is two

```
-BfaGapLimitBits2
```

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

`-BfaTSRon`

Defines

`__BITFIELD_TYPE_SIZE_REDUCTION__`
`__BITFIELD_NO_TYPE_SIZE_REDUCTION__`

Pragmas

None

Description

This option is configurable 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 5.7](#) and [Listing 5.8](#) demonstrate the effects of `-BfaTSRoff` and `-BfaTSRon`, respectively.

Listing 5.7 -BfaTSRoff

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

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

Compiler Options

Compiler Option Details

Listing 5.8 -BfaTSRon

```

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

```

Example

```
-BfaTSRon
```

See also

[Bitfield Type Reduction](#)

-C++ (-C++f, -C++e, -C++c): C++ Support

Group

LANGUAGE

Scope

Compilation Unit

Syntax

```
-C++ (f|e|c)
```

Arguments

f : Full ANSI Draft C++ support

e : Embedded C++ support (EC++)

c : compactC++ support (cC++)

Default

None

Defines

__cplusplus

Pragmas

None

Description

With this option enabled, the Compiler behaves as a C++ Compiler. You can choose between three different types of C++:

- Full ANSI Draft C++ supports the whole C++ language.
- Embedded C++ (EC++) supports a constant subset of the C++ language. EC++ does not support inefficient things like templates, multiple inheritance, virtual base classes and exception handling.
- compactC++ (cC++) supports a configurable subset of the C++ language. You can configure this subset with the option `-Cn`.

If the option is not set, the Compiler behaves as an ANSI-C Compiler.

If the option is enabled and the source file name extension is `*.c`, the Compiler behaves as a C++ Compiler.

If the option is not set, but the source filename extension is `.cpp` or `.cxx`, the Compiler behaves as if the `-C++f` option were set.

Example

```
COMPOPTIONS=-C++f
```

See Also

[-Cn: Disable compactC++ features](#)

-Cc: Allocate Constant Objects into ROM**Group**

OUTPUT

Scope

Compilation Unit

Syntax`-Cc`

Compiler Options

Compiler Option Details

Arguments

None

Default

None

Defines

None

Pragmas

[#pragma INTO_ROM: Put Next Variable Definition into ROM](#)

Description

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 (see the *Linker* manual).

With #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 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 of the Build Tools manual for more information).

NOTE 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

See also

[Segmentation](#)

Linker section in the Build Tools manual

[-F \(-F2, -F2o\): 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 of 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 5.9](#) for a `prm` file describing the placement of sections mentioned in [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 the CodeWarrior IDE refer to the technical note TN234. [Table 5.6](#) indicates how space modifiers are mapped for the RS08.

Table 5.6 Cosmic Space modifier mapping for the RS08

Definition	Keyword Mapping
<code>@tiny</code>	ignored
<code>@near</code>	ignored
<code>@far</code>	ignored

See [Listing 5.9](#) for an example of the `-Ccx` compiler option.

Listing 5.9 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, TN234

-Ci: Tri- and Bigraph Support

Group

LANGUAGE

Compiler Options

Compiler Option Details

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

Bigraph Keyword	Token Replaced	Trigraph Keyword	Token Replaced	Additional Keyword	Token Replaced
<%	}	??=	#	and	&&
%>	}	??/	\	and_eq	&=
<:	[??'	^	bitand	&
:>]	??([bitor	
%:	#	??)]	compl	~
%:%:	##	??!		not	!
		??<	{	or	
		??>	}	or_eq	=
		??-	~	xor	^

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

Bigraph Keyword	Token Replaced	Trigraph Keyword	Token Replaced	Additional Keyword	Token Replaced
				xor_eq	^=
				not_eq	!=

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

Example

-Ci

The example in [Listing 5.10](#) shows the use of trigraphs, bigraphs, and the additional keywords with the corresponding normal C source.

Listing 5.10 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) <%
        i or_eq 0x5;
        i = i xor 0x12;
        i xor_eq 99;
    %> else if (i not_eq 5) <%
```

Compiler Options

Compiler Option Details

```

    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;
        }
    }
}

```

-Cn: Disable compactC++ features

Group

LANGUAGE

Scope

Compilation Unit

Syntax

`-Cn [= {Vf|Tpl|Ptm|Mih|Ctr|Cpr}]`

Arguments

`Vf`: Do not allow virtual functions

`Tpl`: Do not allow templates

`Ptm`: Do not allow pointer to member

`Mih`: Do not allow multiple inheritance and virtual base classes

`Ctr`: Do not create compiler defined functions

`Cpr`: Do not allow class parameters and class returns

Default

None

Defines

None

Pragmas

None

Description

If the `-C++c` option is enabled, you can disable the following compactC++ features:

- `Vf` : Virtual functions are not allowed.
 Avoid having virtual tables that consume a lot of memory.
- `Tpl` : Templates are not allowed.
 Avoid having many generated functions perform similar operations.

Compiler Options

Compiler Option Details

- `Ptm` : Pointer to member not allowed.
Avoid having pointer-to-member objects that consume a lot of memory.
- `Mih` : Multiple inheritance is not allowed.
Avoid having complex class hierarchies. Because virtual base classes are logical only when used with multiple inheritance, they are also not allowed.
- `Ctr` : The C++ Compiler can generate several kinds of functions, if necessary:
 - Default Constructor
 - Copy Constructor
 - Destructor
 - Assignment operator

With this option enabled, the Compiler does not create those functions. This is useful when compiling C sources with the C++ Compiler, assuming you do not want C structures to acquire member functions.
- `Cpr` : Class parameters and class returns are not allowed.
Avoid overhead with Copy Constructor and Destructor calls when passing parameters, and passing return values of class type.

Example

```
-C++c -Cn=Ctr
```

-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:

- 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 5.11](#)) be extended to int. The `-Cni` option disables this extension and saves additional RAM.

Example

See [Listing 5.11](#) for an example of “no integer promotion.”

Listing 5.11 Definition of an ‘old style function’ using a char parameter.

```
old_style_func (a, b, c)
    char a, b, c;
```

Compiler Options

Compiler Option Details

```
{
    ...
}
```

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`, etc.) are treated as integer. If you want them treated as characters, they must be cast ([Listing 5.12](#)).

Listing 5.12 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 5.13](#)).

Listing 5.13 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](#)

-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 its members are `constant` or `volatile`, respectively. Consider the following example.

Example

The source code in [Listing 5.14](#) declares two structs, each of which has a `const` member.

Listing 5.14 Be careful to not write to a constant struct

```
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 causes an error message.

-CswMaxLF: Maximum Load Factor for Switch Tables

Group

CODE GENERATION

Scope

Function

Syntax

`-CswMaxLF<number>`

Arguments

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

Default

Backend-dependent

Defines

None

Pragmas

None

Compiler Options

Compiler Option Details

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

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

The table in [Listing 5.15](#) is filled to 90% (labels for ‘0’ to ‘9’, except for ‘5’).

Listing 5.15 Load factor example

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

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% produces 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](#)
 - [-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

Compiler Options

Compiler Option Details

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.

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:

The table in [Listing 5.16](#) is filled to 90% (labels for ‘0’ to ‘9’, except for ‘5’).

Listing 5.16 Load factor example

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

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.

Compiler Options

Compiler Option Details

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 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.
- 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 5.17 for Loop

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

```
j += i;
}
```

When the `-Cu` compiler option is used, the Compiler issues an information message *Unrolling loop* and transforms this loop as shown in [Listing 5.18](#):

Listing 5.18 Transformation of the for Loop in [Listing 5.17](#)

```
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 5.19 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, because the loop body is never executed:

```
i=1;
```

Listing 5.20 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.

Compiler Options

Compiler Option Details

-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 performs 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 <a 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.

```
-DDEBUG=0
```

This is the same as writing:

```
#define DEBUG 0
```

in the source file.

If you need strings with blanks in your macro definition, there are two ways. Either use escape sequences 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.

Defines

None

Pragmas

None

Examples

See [Listing 5.21](#) and [Listing 5.22](#) for examples using -Ec conversions.

Listing 5.21 Conversion from 'const T*' to 'T*'

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

```
struct A {
    int i;
};

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

void h() {
    const int *i;
    *i=23; /* ANSI-C/C++ illegal, but OK with -Ec! */
}
```

Listing 5.22 Assigning a value to a “constant” pointer

```
-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%e`. A file named `fun.c` creates an encrypted file named `fun.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

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

This encrypts the `fun.c` file using the 1234567 key to the `fun.ec` file, and the `fun.h` file to the `fun.eh` file.

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

```
fun.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

Compiler Options

Compiler Option Details

Example

```
fun.c -Ekey1234567 -Encrypt=%n.e%e
```

This encrypts the fun.' file using the 1234567 key.

See also

[-Encrypt: 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 using filenames with spaces:

```
-Env"OBJPATH=\program files"
```

See also

[Environment](#)

-F (-F2, -F2o): Object-File Format**Group**

OUTPUT

Scope

Application

Syntax

```
-F (2 | 2o)
```

Arguments

2: ELF/DWARF 2.0 object-file format

2o: compatible ELF/DWARF 2.0 object-file format

NOTE Not all object-file formats may be available for a target.

Default

-F2

Defines

```
__ELF_OBJECT_FILE_FORMAT__
```

Compiler Options

Compiler Option Details

Pragmas

None

Description

The Compiler writes the code and debugging info after compilation into an object file. The Compiler produces an ELF/DWARF object file when the `-F2` option is 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.

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.

-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`.

Do not specify any other option or source file when you invoke the `-H` option.

Example

[Listing 5.23](#) lists the short list options.

Listing 5.23 Short Help options

```
-H may produce the following list:  
INPUT:  
-!      Filenames are clipped to DOS length  
-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>`

Compiler Options

Compiler Option Details

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 (`#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](#)

-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 [TEXT_PATH: 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
```

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
```

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

Compiler Options

Compiler Option Details

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 the layout shown in [Listing 5.24](#). The letters in brackets ([]) indicate which suboption may be used to switch it off:

Listing 5.24 -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 [TEXT_PATH: Text File Path](#) environment variable. The defines written to this file depend on the actual Compiler option settings (e.g., type size settings or 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.

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 5.25](#) is an example which lists the contents of a file containing define directives.

Listing 5.25 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__ */
```

Compiler Options

Compiler Option Details

```
#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 5.26](#) 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 5.26 Display contents of a file when include directives are present

```
-Li
If the source file is: C:\myFiles\b.c:
/* 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\hides.h \  
C:\Freescale\lib\targetc\include\stddef.h \  
C:\Freescale\lib\targetc\include\stdtypes.h
```

See also

[-Lm: List of Included Files in Make Format](#)

-Lic: License Information

Group

VARIOUS

Scope

None

Compiler Options

Compiler Option Details

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. Each 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

Compiler Options

Compiler Option Details

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.

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-2005:18:25
```

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`

Compiler Options

Compiler Option Details

See also

- [-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), memory 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 5.27](#) is an example where the use of the `-L1` compiler options allows statistical information to be added to the end of an output listing file.

Listing 5.27 Statistical information appended to an assembler listing

```
-Ll=mylog.txt
/* fun.c */
int Func1(int b) {
    int a = b+3;
    return a+2;
}
void Func2(void) {
}
```

Appends the following two lines into `mylog.txt`:

```
fun.c Func1 -Ll=mylog.txt 11 4 0.055000
fun.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

Compiler Options

Compiler Option Details

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 `make.txt` file, if no argument is 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 5.28](#) is an example where the `-Lm` option generates a make file containing include directives.

Listing 5.28 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
second.o : second.c inc.h header.h
```

See also

[-Li: List of Included Files](#)

[-Lo: Object File List](#)

-LmCfg: Configuration of List of Included Files in Make Format

Group

OUTPUT

Scope

Compilation Unit

Syntax

```
-LmCf $\grave{g}$  [= { i | l | m | n | o | u | q } ]
```

Arguments

- i: Write path of included files
- l: Use line continuation
- m: Write path of main file
- n: No string concatenation
- o: Write path of object file
- u: Update information
- q: Handle single quote (`) as normal token

Default

None

Defines

None

Pragmas

None

Description

This option is used when configuring the [-Lm: List of Included Files in Make Format](#) option. The `-LmCf \grave{g}` 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 in [Listing 5.29](#):

Compiler Options

Compiler Option Details

Listing 5.29 Dependency information grouping

```
<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 `l` suboption uses line continuation for each single entry in the dependency list. This improves readability as shown in [Listing 5.30](#):

Listing 5.30 `l` suboption

```
b.o:      \
  b.c     \
  stdio.h \
  stddef.h \
  stdarg.h \
  string.h
```

With the `m` suboption, 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 `o` suboption 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 `i` suboption writes the full path of all included files in the dependency list ([Listing 5.31](#)):

Listing 5.31 `i` suboption

```
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 `u` suboption 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
```

See also**Compiler options:**

- [-Li: List of Included Files](#)
- [-Lo: Object File List](#)
- [-Lm: List of Included Files in Make Format](#)

-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

Compiler Options

Compiler Option Details

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` file, if no argument given). If the [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

Compiler options:

- [-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

Compiler Options

Compiler Option Details

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

Argument	on	off
c	<code>#line 1</code> <code>#line 10</code>	<code>/* 1 */</code> <code>/* 2 */</code> <code>/* 10 */</code>
e	<code>int j;</code> <code>int i;</code>	<code>int j;</code> <code>int i;</code>
f	<code>C:\Freescale\include\stdlib.h</code>	<code>stdlib.h</code>
l	<code>#line 1 "stdlib.h"</code>	<code>***** FILE 'stdlib.h' */</code>
m		<code>***** FILE 'stdlib.h' */</code>
s	<code>/* 1 */ int f(void) {</code> <code>/* 2 */ return 1;</code> <code>/* 3 */ }</code>	<code>/* 1 */ int f (void) {</code> <code>/* 2 */ return 1 ;</code> <code>/* 3 */ }</code>
all	<code>#line 1</code> <code>"C:\Freescale\include\stdlib.h"</code> <code>#line 10</code>	<code>***** FILE 'stdlib.h' */</code> <code>/* 1 */</code> <code>/* 2 */</code> <code>/* 10 */</code>

Example

```
-LpCfG
-LpCfG=lfs
```

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](#)

-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 (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 appears.

-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 \(-F2, -F2o\): Object-File Format](#)
- [-NoPath: Strip Path Info](#)

-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](#)

-Oa: Alias Analysis Options

Group

OPTIMIZATIONS

Scope

Function

Syntax

`-Oa (addr|ANSI|type|none)`

Arguments

`addr`: All objects in same address area may overlap (safe mode, default)

`ANSI`: use ANSI99 rules

`type`: only objects in same address area with same type may overlap

`none`: assume no objects do overlap

Default

`addr`

Defines

None

Pragmas

None

Description

These four different options allow the programmer to control the alias behavior of the compiler. The option `-oaaddr` is the default because it is safe for all C programs. Use option `-oaansi` if the source code follows the ANSI C99 alias rules. If objects with different types never overlap in your program, use option `-oatype`. If your program doesn't have aliases at all, use option `-oanone` (or the ICG option `-ona`, which is supported for compatibility reasons).

Examples

`-oaANSI`

-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 replaces slower/smaller code with faster/larger code.

Compiler Options

Compiler Option Details

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`

-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 `*.o` extension. 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.

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 `fibonacci.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 because the `<file>` contains a path.

See also

[OBJPATH: Object File Path](#)

-Obsr: Generate Always Near Calls**Group**

OPTIMIZATIONS

Scope

Function

Syntax

```
-Obsr
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Compiler Options

Compiler Option Details

Description

This option forces the compiler to always generate near calls, i.e. use BSR instruction instead of a JSR in order to reduce code size. Without this option the compiler checks the range of the call to determine if a BSR can be generated instead of a JSR.

Example

```
extern int f(void);
```

```
int g(void) {
    return f();
}
```

Without -Obsr:

```
0000 b700      STA      __OVL_g_p0
0002 45       SHA
0003 b700      STA      __OVL_g_14__PSID_75300004
0005 42       SLA
0006 b701      STA      __OVL_g_14__PSID_75300004:1
   4:   return f();
0008 a600      LDA      #__OVL_g_14__PSID_75300001
000a bd0000    JSR      %FIX16(f)
000d 4e000f    LDX      __OVL_g_p0
0010 4e000e    MOV      __OVL_g_14__PSID_75300001,D[X]
0013 2f       INCX
0014 4e010e    MOV      __OVL_g_14__PSID_75300001:1,D[X]
0017 b600      LDA      __OVL_g_14__PSID_75300004
0019 45       SHA
001a b601      LDA      __OVL_g_14__PSID_75300004:1
001c 42       SLA
   5:   }
001d be       RTS
```

With -Obsr:

```
0000 b700      STA      __OVL_g_p0
0002 45       SHA
0003 b700      STA      __OVL_g_14__PSID_75300004
0005 42       SLA
0006 b701      STA      __OVL_g_14__PSID_75300004:1
   4:   return f();
```

```

0008 a600          LDA          #__OVL_g_14__PSID_75300001
000a ad00          BSR          PART_0_7 (f)
000c 4e000f        LDX          __OVL_g_p0
000f 4e000e        MOV          __OVL_g_14__PSID_75300001,D[X]
0012 2f           INCX
0013 4e010e        MOV          __OVL_g_14__PSID_75300001:1,D[X]
0016 b600          LDA          __OVL_g_14__PSID_75300004
0018 45           SHA
0019 b601          LDA          __OVL_g_14__PSID_75300004:1
001b 42           SLA
      5:   }
001c be           RTS

```

-Od: Disable Mid-Level Optimizations

Group

OPTIMIZATIONS

Scope

Function

Syntax

`-Od [= <option Char> {<option Char>}]`

Arguments

<option Char> is one of the following:

- a : Disable mid level copy propagation
- b : Disable mid level constant propagation
- c : Disable mid level common subexpression elimination (CSE)
- d : Disable mid level removing dead assignments
- e : Disable mid level instruction combination
- f : Disable mid level code motion
- g : Disable mid level loop induction variable elimination

Default

None

Compiler Options

Compiler Option Details

Defines

None

Pragmas

None

Description

The backend of this compiler is based on the second generation intermediate code generator (SICG). All intermediate language and processor independent optimizations (cf. NULLSTONE) are performed by the SICG optimizer using the powerful static single assignment form (SSA form). The optimizations are switched off using `-od`. Currently four optimizations are implemented.

Examples

- `-Od` disables all mid-level optimizations
- `-Od=d` disables removing dead assignments only
- `-Od=c` disables removing dead assignments and CSE

See also

None

-Odb: Disable Mid-Level Branch Optimizations

Group

OPTIMIZATIONS

Scope

Function

Syntax

`-Odb [= <option Char> {<option Char>}]`

Arguments

<option Char> is one of the following:

- a: Disable mid level label rearranging
- b: Disable mid level branch tail merging

c: Disable mid level loop hoisting

Default

None

Defines

None

Pragmas

None

Description

This option disables branch optimizations on the SSA form based on control flows. Label rearranging sorts all labels of the control flow to generate a minimum amount of branches.

Branch tail merging places common code into joining labels, as shown:

```
void fun(void) {void fun(void) {  
if(cond) {if(cond) {  
...  
a = 0;} else {  
} else {...  
...}  
a = 0;a = 0;  
}}  
}
```

Examples

-Odb disables all mid-level branch optimizations

-Odb=b disables only branch tail merging

See also

None

-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 5.32](#):

Listing 5.32 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 Do not use this option to specify options with scope Application or Compilation Unit (such as memory model, float or double format, or object-file format) or options for the whole compilation unit (like inlining or macro definition). 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.

Compiler Options

Compiler Option Details

Example

```
-Odocf="-Or|-Cni"
```

-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 `-Oi=c0` option 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 */
    /* ... */
}
```

The option extension [=c<n>] 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:

b: inline calls to the `strlen()` function

d: inline calls to the `fabs()` or `fabsf()` functions

Compiler Options

Compiler Option Details

e: inline calls to the `memset()` function

f: inline calls to the `memcpy()` function

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")`)
- `abs()` or `fabs()` (e.g., `'f = fabs(f);'`)
- `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 – 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 to 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 32-bit shifts, the arrays `_PowOfTwo_16` and `_PowOfTwo_32` are used. These constant arrays contain the values 1, 2, 4, 8, etc. They are declared in `hidef.h`. This optimization is performed only when optimizing for time.
- `-Oilib` without arguments: optimize calls to all supported library functions.

Example

Compiling the `f()` function with the `-Oilib=a` compiler option (only available for ICG-based backends):

```
void f(void) {
    char *s = strcpy(s, ct);
}
```

This translates in a similar fashion to the following function:

```
void g(void) {
    s2 = s;
    while(*s2++ = *ct++);
}
```

See also

[-O: Inlining](#)

-OnB: Disable Branch Optimizer

Group

OPTIMIZATIONS

Scope

Function

Syntax

`-OnB`

Compiler Options

Compiler Option Details

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

With this option, all low-level branch optimizations are disabled.

Example

```
-OnB
```

See Also

None

-OnBRA: Disable JAL to BRA Optimization

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
-OnBRA
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

If the call distance to a subroutine defined in the same compilation unit is in the range of [-512 – 512], the Compiler replaces a JAL instruction by a BRA instruction to reduce code size. Disable this optimization by specifying the option `-OnBRA` if your linker places code between caller and callee.

Example 1: Branch to Subroutine

```
int f(void) {
    return 1;
}
int g(void) {
    return f();
}
```

With `-OnBRA`:

```
__X_f:
F201 LDL        R2, #1
06F6 JAL        R6
__X_g:
7E1E STW       R6, (R0,-R7)
    5:  return f();
F600 LDL        R6, #RS08_8(__X_f)
AE00 ORH       R6, #RS08_8_H(__X_f)
06F6 JAL        R6
6E1D LDW       R6, (R0,R7+)
06F6 JAL        R6
```

Compiler Options

Compiler Option Details

Without -OnBRA:

```

__X_f:
F201 LDL      R2, #1
    3: }
06F6 JAL      R6
__X_g:
7E1E STW      R6, (R0, -R7)
    5:  return f();
06FA TFR      R6, PC
3C00 BRA      __X_f
6E1D LDW      R6, (R0, R7+)
06F6 JAL      R6

```

Example 2: Conditional Branch to Subroutine

```

int a;
int f(void) {
    return 1;
}
void g(void) {
    if (a != 0) {
        (void)f();
    }
}

```

With -OnBRA:

```

__X_f:
F201 LDL      R2, #1
06F6 JAL      R6
__X_g:
7E1E STW      R6, (R0, -R7)
if (a != 0) {
F200 LDL      R2, #RS08_8(a)
AA00 ORH      R2, #RS08_8_H(a)
4A40 LDW      R2, (R2, #0)
}

```

```

1840 TST      R2
2603 BEQ     *+8      ;abs = 0x00000016
      (void)f();
F600 LDL     R6,  #RS08_8(_X_f)
AE00 ORH    R6,  #RS08_8_H(_X_f)
06F6 JAL    R6

6E1D LDW    R6,  (R0,R7+)
06F6 JAL    R6

Without -OnBRA:
__X_f:
F201 LDL     R2,  #1
06F6 JAL    R6

__X_g:
7E1E STW    R6,  (R0,-R7)
      6:   if (a != 0) {
F200 LDL     R2,  #RS08_8(a)
AA00 ORH    R2,  #RS08_8_H(a)
4A40 LDW    R2,  (R2,#0)
1840 TST    R2
06FA TFR    R6,  PC
2400 BNE    __X_f
      9:   }
6E1D LDW    R6,  (R0,R7+)
06F6 JAL    R6

```

Example

-OnBRA

See also

None

-Onbsr: Disable far to near call optimization

Group

OPTIMIZATIONS

Scope

Function

Syntax

-Onbsr

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

This option disables the JSR to BSR optimization. The compiler checks the range of the call to determine if a BSR can be generated instead of a JSR. If -Onbsr is used this optimization will be disabled.

Example

```
int f(void) {
    return 1;
}
int g(void) {
    return f();
}
```

Without -Onbsr:

```

0000 b700          STA      __OVL_g_p0
0002 45           SHA
0003 b700          STA      __OVL_g_14__PSID_75300005
0005 42           SLA
0006 b701          STA      __OVL_g_14__PSID_75300005:1
    5:   return f();
0008 a600          LDA      #__OVL_g_14__PSID_75300002
000a ad00          BSR      PART_0_7 (f)
000c 4e000f        LDX      __OVL_g_p0
000f 4e000e        MOV      __OVL_g_14__PSID_75300002,D[X]
0012 2f           INCX
0013 4e010e        MOV      __OVL_g_14__PSID_75300002:1,D[X]
0016 b600          LDA      __OVL_g_14__PSID_75300005
0018 45           SHA
0019 b601          LDA      __OVL_g_14__PSID_75300005:1
001b 42           SLA
    6:   }
001c be           RTS

```

With -Onbsr:

n.lst -Onbsr

```

0000 b700          STA      __OVL_g_p0
0002 45           SHA
0003 b700          STA      __OVL_g_14__PSID_75300005
0005 42           SLA
0006 b701          STA      __OVL_g_14__PSID_75300005:1
    5:   return f();
0008 a600          LDA      #__OVL_g_14__PSID_75300002
000a bd0000        JSR      %FIX16 (f)
000d 4e000f        LDX      __OVL_g_p0
0010 4e000e        MOV      __OVL_g_14__PSID_75300002,D[X]
0013 2f           INCX
0014 4e010e        MOV      __OVL_g_14__PSID_75300002:1,D[X]
0017 b600          LDA      __OVL_g_14__PSID_75300005
0019 45           SHA
001a b601          LDA      __OVL_g_14__PSID_75300005:1
001c 42           SLA
    6:   }
001d be           RTS

```

-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.

In the ELF format, the object-file format permits optimization 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, it is not possible to separate initialization with 0 from 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

Compiler Options

Compiler Option Details

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` is 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 unoptimized code.

-Onp: Disable Peephole Optimizer

Group

OPTIMIZATIONS

Scope

Function

Syntax

`-Onp`

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

If `-OnP` is specified, the whole peephole optimizer is disabled. The peephole optimizer removes useless loads and stores and applies pre and post increment addressing if possible.

Example

`-Onp`

See Also

None

-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

Compiler Options

Compiler Option Details

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

```
-On.PMNC
```

-Onr: Disable Reload from Register Optimization

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
-Onr
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

This option disables the low level register trace optimizations. If you use the option the code becomes more readable, but less optimal.

Example

`-Onr`

See Also

None

-Ont: Disable Tree Optimizer

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
-Ont [= {%|&|*|+|-|/|0|1|7|8|9|?|^|a|b|c|d|e|
      f|g|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

Compiler Options

Compiler Option Details

- c: Disable compare optimization
- d: Disable binary operation optimization
- e: Disable constant swap optimization
- f: Disable condition optimization
- g: Disable compare size 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
- 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=l

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)
```

Compiler Options

Compiler Option Details

is 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;
```

is optimized into:

```
uL = uL1 >> (unsigned char)uL2;
```

Optimization of zero shift counts:

```
uL = uL1 >> 0;
```

is optimized into:

```
uL = uL1;
```

Optimization of shift counts greater than the object to be shifted:

```
uL = uL1 >> 40;
```

is optimized into:

```
uL = 0L;
```

Strength reduction for operations followed by a cut operation:

```
ch = uL1 * uL2;
```

is optimized into:

```
ch = (char)uL1 * (char)uL2;
```

Replacing shift operations by load or store

```
i = uL >> 16;
```

is optimized into:

```
i = *(int *) (&uL);
```

Shift count reductions:

```
ch = uL >> 17;
```

is optimized into:

```
ch = (*(char *) (&uL)+1)>>1;
```

Optimization of shift combined with binary and:

```
ch = (uL >> 25) & 0x10;
```

is 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 transforms 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)'

Compiler Options

Compiler Option Details

-Ont=b

Disable optimization of '3+7' into '10'

-Ont=a

Disable optimization of last statement in function if result is not used

-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 -Ont
```

-Ontc: disable tail call optimization**Group**

OPTIMIZATIONS

Scope

Function

Syntax

-Ontc

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

By default, the compiler replaces trailing calls (JSR/BSR) with JMP instructions if the function does not contain any other function calls. This allows the compiler to remove all the entry and exit code from the current function.

Compiler Options

Compiler Option Details

Example

```
void f1() {
    i++;
}
```

```
void f2() {
    f1();
}
```

Without -Ontc:

```
Function: f1
Source  : d:\junk\rs08\test.c
Options : -Lasm=%n.lst
```

```
 7:    i++;
0000 3c01          INC     i:1
0002 3602          BNE    L6
0004 3c00          INC     i
0006          L6:
 8:    }
0006 be          RTS
 9:
10: void f2() {
```

```
Function: f2
Source  : d:\junk\rs08\test.c
Options : -Lasm=%n.lst
```

```
11:    f1();
0000 3000          BRA    PART_0_7(f1)
12:    }
```

With -Ontc:

```
Function: f1
Source  : d:\junk\rs08\test.c
Options : -Lasm=%n.lst -Ontc
```

```

7:      i++;
0000 3c01          INC      i:1
0002 3602          BNE     L6
0004 3c00          INC      i
0006          L6:
8:      }
0006 be          RTS
9:
10: void f2() {

```

Function: f2
Source : d:\junk\rs08\test.c
Options : -Lasm=%n.lst -Ontc

```

0000 45          SHA
0001 b700          STA     __OVL_f2_14__PSID_75300003
0003 42          SLA
0004 b701          STA     __OVL_f2_14__PSID_75300003:1
11:      f1();
0006 ad00          BSR     PART_0_7(f1)
0008 b600          LDA     __OVL_f2_14__PSID_75300003
000a 45          SHA
000b b601          LDA     __OVL_f2_14__PSID_75300003:1
000d 42          SLA
12:      }
000e be          RTS
13:

```

-Ostk: Reuse Locals of Stack Frame

Group

OPTIMIZATIONS

Scope

Function

Syntax

-Ostk

Compiler Options

Compiler Option Details

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

This option instructs the compiler to reuse the location of local variables/temporaries whenever possible. When used, the compiler analyzes which local variables are alive simultaneously. Based on that analysis the compiler chooses the best memory layout for variables. Two or more variables may end up sharing the same memory location.

Example

TBD

-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
```

but used in:

```
printf(STRING);
```

produces a carriage return with line feed:

```
C:  
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.

Compiler Options

Compiler Option Details

This occurs because the `STRING` macro has to be the same for both 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 through z or A through 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 in [Listing 5.33](#):

Listing 5.33 Conditional compilation

```
/* Header file myfile.h */
#ifndef _MY_FILE_H_
#define _MY_FILE_H_

/* ... content ... */
#endif /* _MY_FILE_H_ */
```

When the `#ifndef` and `#define` directives are issued, any header file content is read only once even when the header file is included several times. This solves many 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 in this manner, 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 that do not contain macros of that nature.

Example

-Pio

-Prod: Specify Project File at Startup

Group

Startup - This option cannot be specified interactively.

Compiler Options

Compiler Option Details

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
```

NOTE 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 | 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 Specifying an unsupported qualifier has no effect, 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).

-T: Flexible Type Management

Group

LANGUAGE.

Scope

Application

Syntax

-T<Type Format>

Arguments

<Type Format>: See below

Default

Depends on target, see the Backend chapter

Defines

To deal with different type sizes, one of the following define groups in [Listing 5.34](#) is predefined by the Compiler:

Listing 5.34 Predefined 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__
__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_DSP__

__DOUBLE_IS_IEEE32__
__DOUBLE_IS_DSP__

__LONG_DOUBLE_IS_IEEE32__
__LONG_DOUBLE_IS_DSP__

__LONG_LONG_DOUBLE_IS_IEEE32__
__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

Compiler Options

Compiler Option Details

Description

This option allows configurable type settings. The option syntax 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` or `F` may be used for the type `float`.

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
DSP (32-bit)	0

Not all formats may be available for a target. See [RS08 Backend](#) for supported formats.

Compiler Options

Compiler Option Details

NOTE At least one type for each basic size (1, 2, 4 bytes) has to be available. It is illegal if no type of any sort is not set to at least a size of one. See [RS08 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.

Examples

```
-Tsc sets 'char' to 'signed char'
and
-Tuc sets 'char' to 'unsigned char'
```

Listing 5.35 -Tsc1s2i2L4LL4f2e2 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)
enum with 16 bits (signed) (e2)
```

For integrity and compliance to ANSI, the following two rules must be true:

Listing 5.36 Restrictions

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

NOTE It is not permitted to set `char` to 16 bits and `int` to 8 bits.

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; /* -Til 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
```

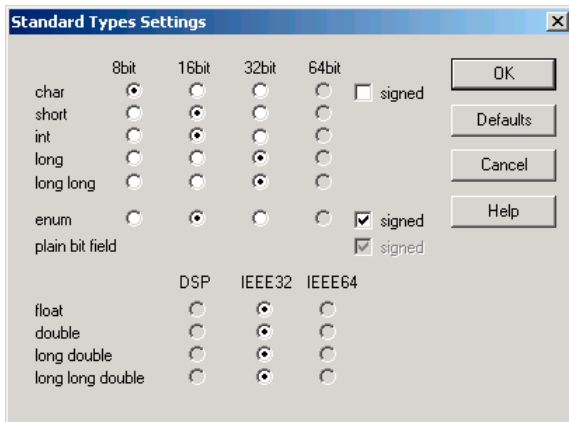
Setting float to IEEE32:

```
-Tf2
```

The `-TvtD` 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 ([Figure 5.3](#)):

Figure 5.3 Standard Types Settings dialog box



Compiler Options

Compiler Option Details

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

Compiler Options

Compiler Option Details

Description

The application starts 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
```

Compiler Options

Compiler Option Details

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

Some editors (e.g., early versions of WinEdit) expect the filename in the Microsoft message format (8.3 format). That means the filename can have, at most, eight characters with not more than a three-character extension. Longer filenames are possible when you use later versions of Windows. This option truncates the filename to the 8.3 format.

Example

```
x:\mysourcefile.c(3): INFORMATION C2901: Unrolling loop
```

With the `-Wmsg8x3` option 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

Compiler Options

Compiler Option Details

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.

Compiler Options

Compiler Option Details

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 appropriate argument from an external editor), 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 the examples in [Listing 5.37](#)).

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.

Compiler Options

Compiler Option Details

NOTE Using the verbose message format may slow down the compilation because the compiler has to write more information into the message file.

Example

See [Listing 5.37](#) for examples showing the differing message formats.

Listing 5.37 Message file formats (batch mode)

```
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

See [Listing 5.38](#) for examples showing the differing message formats.

Compiler Options

Compiler Option Details

Listing 5.38 Message file formats (interactive mode)

```
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%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

Format	Description	Example
%s	Source Extract	
%p	Path	X:\Freescale\
%f	Path and name	X:\Freescale\mysourcefile
%n	filename	mysourcefile
%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

Compiler Options

Compiler Option Details

Table 5.13 Message Format Specifiers (*continued*)

Format	Description	Example
%m	Message	text
%%	Percent	%
\n	New line	
% "	A " if the filename, path, or extension contains a space	
% '	A ' if the filename, path, or 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

Compiler Options

Compiler Option Details

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

Format	Description	Example
%s	Source Extract	
%p	Path	X:\sources\
%f	Path and name	X:\sources\mysourcefile
%n	filename	mysourcefile
%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, path, or extension contains a space.	
%'	A ' if the filename, path, or 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 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

Format	Description	Example
%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

```
-WmsgFonf "%k %d: %m\n"
```

Produces a message in following format:

```
information L10324: Linking successful
```

See also

[ERRORFILE: Error filename Specification](#)

Compiler options:

- [-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

Format	Description	Example
<code>%K</code>	Uppercase kind	ERROR
<code>%k</code>	Lowercase kind	error
<code>%d</code>	Number	C1815
<code>%m</code>	Message	text

Compiler Options

Compiler Option Details

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](#)

Compiler options:

- [-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

Compiler Options

Compiler Option Details

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

```
-WmsgNi10
```

Ten information messages logged

See also

Compiler options:

- [-WmsgNe: Number of Error Messages](#)
- [-WmsgNw: Number of Warning Messages](#)

-WmsgNu: Disable User Messages

Group

MESSAGES

Scope

None

Syntax

```
-WmsgNu [= { a | b | c | d } ]
```

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).
- 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

Compiler options:

- [-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](#).

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 to create an error listing file. 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 environment variable [ERRORFILE: Error filename Specification](#).

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 issues 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;
    func(a, b); // <- Error here - only two parameters
}
func(a, b, c)
    char a, b, c;
{
    ...
}
```

See also

[-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

Compiler Options

Compiler Option Details

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`

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

Compiler Options

Compiler Option Details

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 -Ansi: Strict ANSI compiler option 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. It is also possible to log all Compiler predefined defines to a file using the [-Ldf: Log Predefined Defines to File](#) compiler option.

Compiler Predefined Macros

Compiler Vendor Defines

Compiler Vendor Defines

[Table 6.2](#) shows the defines identifying the Compiler vendor. Compilers in the USA may also be sold by ARCHIMEDES.

Table 6.2 Compiler Vendor Identification Defines

Name	Defined
__HIWARE__	always
__MWERKS__	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
__PRODUCT_HICROSS_PLUS__	defined for V5.0 Compilers
__DEMO_MODE__	defined if the Compiler is running in demo mode
__VERSION__	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 Compiler provides the “endian” macros listed in [Table 6.4](#).

Table 6.4 Compiler macros for defining “endianness”

Name	Defined
__LITTLE_ENDIAN__	defined if the Compiler allocates in Little Endian order
__BIG_ENDIAN__	defined if the Compiler allocates in Big Endian order

The following example illustrates the differences between little endian and big endian ([Listing 6.1](#)).

Listing 6.1 Little vs. big endian

```
unsigned long L = 0x87654321;
unsigned short s = *(unsigned short*)&L; // BE: 0x8765, LE: 0x4321
unsigned char c = *(unsigned 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")) {
        i=2;
    }
}
```

You can check all preprocessor-valid 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 OPTIONS`.

As a parameter, only the option itself is tested and not a specific argument of an option.

For example:

```
#if __OPTION_ACTIVE__("-D") /* true if any -d option given
*/
#if __OPTION_ACTIVE__("-DABS") /* 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 6.3](#) show part of the contents of `stdtypes.h` (included from `stddef.h`).

Listing 6.3 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
    #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

```

Compiler Predefined Macros

ANSI-C Standard Types 'size_t', 'wchar_t' and 'ptrdiff_t' Defines

[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.
<code>__WCHAR_T_IS_UCHAR__</code>	Defined if the Compiler expects <code>wchar_t</code> in <code>stddef.h</code> to be unsigned char.
<code>__WCHAR_T_IS_USHORT__</code>	Defined if the Compiler expects <code>wchar_t</code> in <code>stddef.h</code> to be unsigned short.
<code>__WCHAR_T_IS_UINT__</code>	Defined if the Compiler expects <code>wchar_t</code> in <code>stddef.h</code> to be unsigned int.
<code>__WCHAR_T_IS_ULONG__</code>	Defined if the Compiler expects <code>wchar_t</code> in <code>stddef.h</code> to be unsigned long.
<code>__PTRDIFF_T_IS_CHAR__</code>	Defined if the Compiler expects <code>ptrdiff_t</code> in <code>stddef.h</code> to be char.
<code>__PTRDIFF_T_IS_SHORT__</code>	Defined if the Compiler expects <code>ptrdiff_t</code> in <code>stddef.h</code> to be short.
<code>__PTRDIFF_T_IS_INT__</code>	Defined if the Compiler expects <code>ptrdiff_t</code> in <code>stddef.h</code> to be int.
<code>__PTRDIFF_T_IS_LONG__</code>	Defined if the Compiler expects <code>ptrdiff_t</code> in <code>stddef.h</code> to be long.

The following tables show the default settings of the ANSI-C Compiler `size_t` and `ptrdiff_t` standard types.

Macros for RS08

[Table 6.7](#) shows the settings for the RS08 target:

Table 6.7 RS08 Compiler Defines

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 RS08 Compiler Pointer Difference Macros

ptrdiff_t Macro	Defined
__PTRDIFF_T_IS_CHAR__	never
__PTRDIFF_T_IS_SHORT__	never
__PTRDIFF_T_IS_INT__	always
__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 6.4](#).

Listing 6.4 Effect of polarity upon division and modulus arithmetic.

```

#ifdef __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

```

Compiler Predefined Macros

Object-File Format Defines

```
-22 / 7 == -3;  -22 % 7 == -1
-22 /-7 == 3;  -22 % -7 == -1
#endif
```

The following sections show how it is implemented in a backend.

Macros for RS08

Table 6.9 RS08 Compiler Modulo Operator Macros

Name	Defined
__MODULO_IS_POSITIV__	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
__ELF_OBJECT_FILE_FORMAT__	-F2

Bitfield Defines

This section describes the defines and define groups available for the RS08 compiler.

Bitfield Allocation

The Compiler provides six predefined macros to distinguish between the different allocations:

```
__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 above-listed defines, 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 6.5](#)).

Listing 6.5 Compatible bitfield allocation

```

struct {
    /* Memory layout of I/O port:

        name:          MSB | CCR | DIR | DATA | LSB
        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 in [Listing 6.6](#):

Listing 6.6 `__BITFIELD_MSBYTE_FIRST__` definition

```

/* example for __BITFIELD_MSBYTE_FIRST__ */
struct {

```

Compiler Predefined Macros

Bitfield Defines

```

unsigned char a:8;
unsigned char b:3;
unsigned char c:5;
} MyIOport2;

/* LSBIT_FIRST          */ /* MSBIT_FIRST          */
/* MSByte  LSByte      */ /* MSByte  LSByte      */
/* aaaaaaaaa cccccbbb */ /* aaaaaaaaa bbbcccc   */

```

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 to is non-portable and, for the masking involved in unpacking individual fields, inefficient. 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. 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.

```

__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 6.7](#)):

Listing 6.7 Compatible optimized bitfields

```

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 in the following ([Listing 6.8](#)):

Listing 6.8 Plain bitfield

```
struct _bits {
    int myBits:3;
} bits;
```

`myBits` is signed, because plain `int` is also signed. To implement it as an unsigned bitfield, use the following code ([Listing 6.9](#)):

Listing 6.9 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:

```
define group__PLAIN_BITFIELD_IS_SIGNED__ /* defined if plain
bitfield
__PLAIN_BITFIELD_IS_UNSIGNED__ is signed */
/* defined if plain bitfield
is unsigned */
```

Macros for RS08

[Table 6.11](#) identifies the implementation in the Backend.

Table 6.11 RS08 Compiler—Backend Macros

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 -T option or Backend
__CHAR_IS_UNSIGNED__	see -T option or Backend
__CHAR_IS_8BIT__	see -T option or Backend
__CHAR_IS_16BIT__	see -T option or Backend
__CHAR_IS_32BIT__	see -T option or Backend
__CHAR_IS_64BIT__	see -T option or Backend

Table 6.12 Type Information Defines (continued)

Name	Defined
__SHORT_IS_8BIT__	see -T option or Backend
__SHORT_IS_16BIT__	see -T option or Backend
__SHORT_IS_32BIT__	see -T option or Backend
__SHORT_IS_64BIT__	see -T option or Backend
__INT_IS_8BIT__	see -T option or Backend
__INT_IS_16BIT__	see -T option or Backend
__INT_IS_32BIT__	see -T option or Backend
__INT_IS_64BIT__	see -T option or Backend
__ENUM_IS_8BIT__	see -T option or Backend
__ENUM_IS_SIGNED__	see -T option or Backend
__ENUM_IS_UNSIGNED__	see -T option or Backend
__ENUM_IS_16BIT__	see -T option or Backend
__ENUM_IS_32BIT__	see -T option or Backend
__ENUM_IS_64BIT__	see -T option or Backend
__LONG_IS_8BIT__	see -T option or Backend
__LONG_IS_16BIT__	see -T option or Backend
__LONG_IS_32BIT__	see -T option or Backend
__LONG_IS_64BIT__	see -T option or Backend
__LONG_LONG_IS_8BIT__	see -T option or Backend
__LONG_LONG_IS_16BIT__	see -T option or Backend
__LONG_LONG_IS_32BIT__	see -T option or Backend
__LONG_LONG_IS_64BIT__	see -T option or Backend
__FLOAT_IS_IEEE32__	see -T option or Backend
__FLOAT_IS_DSP__	see -T option or Backend
__DOUBLE_IS_IEEE32__	see -T option or Backend

Compiler Predefined Macros

Bitfield Defines

Table 6.12 Type Information Defines (continued)

Name	Defined
__DOUBLE_IS_DSP__	see -T option or Backend
__LONG_DOUBLE_IS_IEEE32__	see -T option or Backend
__LONG_DOUBLE_IS_DSP__	see -T option or Backend
__LONG_LONG_DOUBLE_IS_IEEE32__	see -T option or Backend
__LONG_LONG_DOUBLE_IS_DSP__	see -T option or Backend
__VTAB_DELTA_IS_8BIT__	see -T option
__VTAB_DELTA_IS_16BIT__	see -T option
__VTAB_DELTA_IS_32BIT__	see -T option
__VTAB_DELTA_IS_64BIT__	see -T option
__PLAIN_BITFIELD_IS_SIGNED__	see option -T or Backend
__PLAIN_BITFIELD_IS_UNSIGNED__	see option -T or Backend

Freescale RS08-Specific Defines

[Table 6.13](#) identifies implementations specific to the Backend.

Table 6.13 RS08 Back End Defines

Name	Defined
__RS08__	always
__NO_RECURSION__	always
__PTR_SIZE_1__	always
__PTR_SIZE_2__	never
__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 one file.

Listing 7.1 Pragma syntax

```
#pragma pragma_name [optional_arguments]
```

The value for `optional_arguments` 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.

Compiler Pragmas

Pragma Details

Table 7.1 Pragma documentation topics

Topic	Description
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 for pragmas.

Table 7.2 Definition of items that can appear in a pragma's scope topic

Scope	Description
File	The pragma is valid from the current position until the end of the source file. For 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. For 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 CONST_SEG: Constant Data Segment Definition

Scope

Until the next CONST_SEG pragma

Syntax

```
#pragma CONST_SEG (<Modif> <Name> | DEFAULT)
```

Synonym

CONST_SECTION

Arguments

Listing 7.2 Some strings which may be used for <Modif>

```
__FAR_SEG      (compatibility alias: FAR)  
__PAGED_SEG
```

NOTE Do not use a compatibility alias 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. The `__SHORT_SEG` modifier specifies a segment which is accessed with 8-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. Refer to the linker section of the Build Tools Utilities manual for details.

Default

DEFAULT

Description

This pragma allocates constant variables into a segment. The segment is then allocated in the link parameter file to specific addresses. The CONST_SEG pragma

Compiler Pragmas

Pragma Details

sets the current `const` segment. All constant data declarations are placed in this segment. The default segment is set with:

```
#pragma CONST_SEG DEFAULT
```

With the `-Cc` option set, constants are always allocated in constant segments in the ELF object-file format and after the first `#pragma CONST_SEG` (see [-Cc: Allocate Constant Objects into ROM](#)).

The `CONST_SEG` pragma also affects constant data declarations as well as definitions. Ensure that all constant data 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 `CONST_SECTION` synonym has exactly the same meaning as `CONST_SEG`.

Examples

[Listing 7.3](#) shows code that uses the `CONST_SEG` pragma.

Listing 7.3 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;
}

/* 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 7.4](#) shows code that uses the CONST_SEG pragma *improperly*.

Listing 7.4 Improper use of the CONST_SEG pragma

```
#pragma DATA_SEG CONST1
#pragma CONST_SEG CONST1 /* error: same 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 different segments */

#pragma CONST_SEG __SHORT_SEG DEFAULT /* error: no modifiers for the
                                       DEFAULT segment are allowed
```

See also

[RS08 Backend](#)

Linker section of the Build Tools manual

[#pragma DATA_SEG: Data Segment Definition](#)

[#pragma STRING_SEG: String Segment Definition](#)

[#pragma INTO_ROM: Put Next Variable Definition into ROM](#)

[-Cc: Allocate Constant Objects into ROM](#)

Compiler Pragmas

Pragma Details

#pragma CREATE_ASM_LISTING: Create an Assembler Include File Listing

Scope

Until the next CREATE_ASM_LISTING pragma

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 determines if the following defines or objects are printed into the assembler include file.

A new file is generated only when the `-La` compiler option is specified together with a header file containing `#pragma CREATE_ASM_LISTING ON`.

Listing 7.5 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

[Generating Assembler Include Files \(-La Compiler Option\)](#)

#pragma DATA_SEG: Data Segment Definition

Scope

Until the next DATA_SEG pragma

Syntax

```
#pragma DATA_SEG (<Modif> <Name> | DEFAULT)
```

Synonym

DATA_SECTION

Arguments

Listing 7.6 Some of the strings which may be used for <Modif>

```
__TINY_SEG
__SHORT_SEG (compatibility alias: SHORT)
__DIRECT_SEG (compatibility alias: DIRECT)
__PAGED_SEG
__FAR_SEG (compatibility alias: FAR)
```

NOTE Do not use a compatibility alias 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 problems which may be hard to detect. So avoid using compatibility alias names.

`__TINY_SEG`: specifies an operand that can be encoded on four bits (between 0x0 and 0xF). Use this modifier for frequently accessed global variables.

`__SHORT_SEG`: specifies an operand that can be encoded on five bits (between 0x0 and 0x1F). Use this modifier to access IO registers in the lower RS08 register bank.

`__DIRECT_SEG`: specifies an operand that can be encoded within the range 0x00 and 0xBF. Use this modifier to access global variables. If the operand does not fit into either four bits (`__TINY_SEG`) or five bits (`__SHORT_SEG`) then direct (8 bit) addressing is used.

`__PAGED_SEG`: specifies an operand that can be encoded within the range 0x100 and 0x3FF for read/write registers and within the range 0x00 and 0x3FFF for global read-only data. Use this modifier to access IO registers in the high RS08

Compiler Pragmas

Pragma Details

register bank (read/write) or to access constant data. Objects allocated using `___PAGED_SEG` must not cross page boundaries.

`___FAR_SEG`: specifies an operand that can be encoded within the range 0x00 to 0x3FFF. Use this modifier to access large constant (read-only) data. Allocate FAR sections to multiple pages.

<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. Refer to the linker manual for details.

Default

DEFAULT

Description

The `DATA_SEG` pragma allocates variables into the current data segment. This segment is used to place all variable declarations. Use this pragma to impose tiny or short addressing mode when accessing variables in the relevant sections. Set the default segment with:

```
#pragma DATA_SEG DEFAULT
```

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.

The RS08 compiler automatically allocates non-static local data into an OVERLAP section. The OVERLAP section uses the same address range as `___DIRECT_SEG` (0x00 – 0xBF).

Some instructions support tiny and short addressing. These instructions are encoded on one byte only rather than two bytes.

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 to 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 synonym `DATA_SECTION` means exactly same as `DATA_SEG`.

Example

[Listing 7.7](#) shows source code that uses the `DATA_SEG` pragma.

Listing 7.7 Using the DATA_SEG pragma

```
/* in a header file */
#pragma DATA_SEG __TINY_SEG MyTinySection
char status;

#pragma DATA_SEG __SHORT_SEG MyShortSection
unsigned char IOReg;

#pragma DATA_SEG DEFAULT
char temp;

#pragma DATA_SEG __PAGED_SEG MyShortSection
unsigned char IOReg;
unsigned char *__paged io_ptr = &IOREG;

#pragma DATA_SEG __PAGED_SEG MyPagedSection
const char table[10];
unsigned char *__paged tblptr = table;

#pragma DATA_SEG __FAR_SEG MyFarSection
const char table[1000];
unsigned char *__far tblptr = table;
```

See also

[RS08 Backend](#)

Linker section of the Build Tools manual

[#pragma CONST_SEG: Constant Data Segment Definition](#)

[#pragma STRING_SEG: String Segment Definition](#)

[#pragma INTO_ROM: Put Next Variable Definition into ROM](#)

[-Cc: Allocate Constant Objects into ROM](#)

#pragma INLINE: Inline Next Function Definition**Scope**

Function Definition

Compiler Pragmas

Pragma Details

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 7.8 Using an INLINE pragma to inline a function

```
int i;
#pragma INLINE
static void fun(void) {
    i = 12;
}
void main(void) {
    fun(); // results in inlining 'i = 12;'
}
```

See also

[#pragma NO_INLINE: Do not Inline next function definition](#)

[-Oi: Inlining](#)

#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 subsequent segment pragma (`CONST_SEG`, `DATA_SEG`, `CODE_SEG`) disables the pragma.

NOTE This pragma is only useful for the HIWARE object-file format (but not for ELF/DWARF).

NOTE This pragma is to force a non-constant (meaning a 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 7.9](#) presents some examples which use the `INTO_ROM` pragma.

Listing 7.9 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
```

Compiler Pragmas

Pragma Details

```
#pragma DATA_SEG MySeg /* INTO_ROM overwritten! */
int other2; /* put into MySeg */
```

See also

[-Cc: Allocate Constant Objects into ROM](#)

#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 name that is used and its content have no meaning other than each name can contain only one content string. However, multiple pragmas with different NAMES are legal.

For the Linker or for the Debugger, however, NAME might trigger some special functionality with CONTENT as an argument.

The Linker collects the CONTENT for every NAME from different object files and issues a message if CONTENT differs in 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 7.10](#) in a header file used by all compilation units, the Linker issues a message if the object files built with `_DEBUG` are linked with object files built without it.

Listing 7.10 Using pragmas to assist in debugging

```
#ifndef _DEBUG
    #pragma LINK_INFO MY_BUILD_ENV DEBUG
#else
    #pragma LINK_INFO MY_BUILD_ENV NO_DEBUG
#endif
```

#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 setting the `-Cu` option for the following single function.

Compiler Pragmas

Pragma Details

Listing 7.11 Using a LOOP_UNROLL pragma to unroll the 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 - no quote marks needed}
```

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

For the example in [Listing 7.12](#) the pragma accesses declarations and definitions.

Listing 7.12 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;
static int ref;

#pragma mark -
static void inc_counter(void) {
    counter++;
}
static void inc_ref(void) {
    ref++;
}
```

Compiler Pragmas

Pragma Details

#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 messages 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's scope. For example, 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 7.13](#), parentheses () were omitted.

Listing 7.13 Using the MESSAGE Pragma

```
/* treat C1412: Not a function call, */  
/* address of a function, as error */  
#pragma MESSAGE ERROR C1412  
void f(void);  
void main(void) {  
    f; /* () is missing, but still legal in C */  
/* ERROR because of pragma MESSAGE */  
}
```

See also**Compiler options:**

- [-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

Compiler Pragmas

Pragma Details

Description

This pragma suppresses the generation of entry code and is useful for inline assembler functions. The entry code prepares subsequent C code to run properly. It usually consists of pushing register arguments on the stack (if necessary), and allocating the stack space used for local variables and temporaries and storing callee saved registers according to the calling convention.

The main purpose of this pragma is for functions which contain only High-Level Inline (HLI) assembler code to suppress the compiler generated entry code.

One use of this pragma is in the startup function `_Startup`. At the start of this function the stack pointer is not yet defined. It has to be loaded by custom HLI code first.

NOTE C code inside of a function compiled with `#pragma NO_ENTRY` generates independently of this pragma. The resulting C code may not work since it could access unallocated variables in memory.

This pragma is safe in functions with only HLI code. In functions that contain C code, using this pragma is a very advanced topic. Usually this pragma is used together with the pragma `NO_FRAME`.

TIP Use a `#pragma NO_ENTRY` and a `#pragma NO_EXIT` with HLI-only functions to avoid generation of any additional frame instructions by the compiler.

The code generated in a function with `#pragma NO_ENTRY` may be unreliable. It is assumed that the user ensures correct memory use.

WARNING! Not all backends support this pragma. Some may still generate entry code even if this pragma is specified.

Example

[Listing 7.14](#) 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 7.14 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.*/
        ...
    }
```

```
}  
}
```

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 two pragmas NO_ENTRY and NO_EXIT together avoid generation of any exit/entry code. Functions written in High-Level Inline (HLI) assembler can therefore be used as custom entry and exit code.

The compiler can often deduce if a function does not return, but sometimes this is not possible. This pragma can then be used to avoid the generation of exit code.

TIP Use a #pragma NO_ENTRY and a #pragma NO_EXIT with HLI-only functions to avoid generation of any additional frame instructions by the compiler.

Compiler Pragmas

Pragma Details

The code generated in a function with `#pragma NO_EXIT` may not be safe. It is assumed that the user ensures correct memory usage.

NOTE Not all backends support this pragma. Some may still generate exit code even if this pragma is specified.

Example

[Listing 7.15](#) 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 7.15 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.*/
        ...
    }
}
```

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 is accepted for compatibility only. It is replaced by the `#pragma NO_ENTRY` and `#pragma NO_EXIT` pragmas.

For some compilers, using this pragma does not affect the generated code. Use the two pragmas `NO_ENTRY` and `NO_EXIT` instead (or in addition). When the compiler does consider this pragma, see the `#pragma NO_ENTRY` and `#pragma NO_EXIT` for restrictions that apply.

This pragma suppresses the generation of frame code and is useful for inline assembler functions.

The code generated in a function with `#pragma NO_FRAME` may be unreliable. It is assumed that the user ensures correct memory usage.

NOTE Not all backends support this pragma. Some may still generate frame code even if this pragma is specified.

Example

[Listing 7.16](#) 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.

Compiler Pragmas

Pragma Details

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.*/
        ...
    }
}
```

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 from inlining the next function in the source. The pragma is used to avoid inlining a function which would be inlined because of the `-Oi` compiler option.

Listing 7.17 Use of #pragma NO_INLINE to prevent inlining a function.

```
// (With the -Oi option)
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](#)

#pragma NO_LOOP_UNROLL: Disable Loop Unrolling**Scope**

Function

Syntax

```
#pragma NO_LOOP_UNROLL
```

Synonym

None

Arguments

None

Compiler Pragmas

Pragma Details

Default

None

Description

If this pragma is present, no loop unrolling is performed for the next function definition, even if the `-Cu` command line option is given.

Example

Listing 7.18 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](#)

[-Cu: Loop Unrolling](#)

#pragma NO_RETURN: No Return Instruction

Scope

Function

Syntax

```
#pragma NO_RETURN
```

Synonym

None

Arguments

None

Default

None

Description

This pragma suppresses the generation of the return instruction (return from a subroutine or return from an 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 functions into a segment that only has a linear memory area.

Example

The example in [Listing 7.19](#) 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.

Listing 7.19 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) {
    ...
}
```

Compiler Pragmas

Pragma Details

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

None

Arguments

None

Default

None

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 shows the use of the pragma. Without the pragma, # is handled as a string constructor, which is not the desired behavior.

Listing 7.20 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
```

Compiler Pragmas

Pragma Details

See also

[-Pio: Include Files Only Once](#)

#pragma OPTION: Additional Options

Scope

Compilation Unit or until the next OPTION pragma

Syntax

```
#pragma OPTION ADD [<Handle>] "<Option>"
#pragma OPTION DEL <Handle>
#pragma OPTION DEL ALL
```

Synonym

None

Arguments

<Handle>: An identifier - added options can selectively be deleted.

<Option>: A valid option string

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 OPTION ADD pragma.

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.
- These 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.](#) and
 - [-WmsgSw: Setting a Message to Warning.](#)
 Use [#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 7.21](#) shows how to compile only a single function with the additional `-Or` option.

Listing 7.21 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 #pragma OPTION ADD */
/* are active again. */
```

Compiler Pragmas

Pragma Details

The examples in [Listing 7.22](#) show *improper* uses of the OPTION pragma.

Listing 7.22 Improper uses of the OPTION pragma

```
#pragma OPTION ADD -Or /* ERROR, quotes missing; use "-Or" */

#pragma OPTION "-Or" /* ERROR, needs also the ADD keyword */

#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"
#ifdef __CNI__
/* ERROR, macros are not defined for options */
/* added with the pragma */
#endif
```

#pragma REALLOC_OBJ: Object Reallocation

Scope

Compilation Unit

Syntax

```
#pragma REALLOC_OBJ "segment" ["objfile"] object qualifier
```

Arguments

segment: Name of an already existing segment. This name must have been previously used by a segment pragma (DATA_SEG, CODE_SEG, CONST_SEG, or STRING_SEG).

objfile: Name of a object file. If specified, the object is assumed to have static linkage and to be defined in *objfile*. The name must be specified without alteration by the qualifier `__namemangle`.

object: Name of the object to be reallocated. Here the name as known to the Linker has to be specified.

qualifier: One of the following:

- `__near`,
- `__far`,
- `__paged`, or
- `__namemangle`.

Some of the qualifiers are only allowed to backends not supporting a specified qualifier generating this message. With the special `__namemangle` qualifier, the link name is changed so that the name of the reallocated object does not match the usual name. This feature detects when a `REALLOC_OBJ` pragma is not applied to all uses of one object.

Default

None

Description

This pragma reallocates an object (e.g., affecting its calling convention). This is used by the linker if the linker has to distribute objects over banks or segments in an automatic way (code distribution). The linker is able to generate an include file containing `#pragma REALLOC_OBJ` to tell the compiler how to change calling conventions for each object. See the Linker manual for details.

Example

[Listing 7.23](#) uses the `REALLOC_OBJ` pragma to reallocate the `evaluate.o` object file.

Listing 7.23 Using the `REALLOC_OBJ` pragma to reallocate an object

```
#pragma REALLOC_OBJ "DISTRIBUTE1" ("evaluate.o") Eval_Plus __near  
__namemangle
```

See also

Message C420 in the Online Help

Linker section of the Build Tools manual

Compiler Pragmas

Pragma Details

#pragma STRING_SEG: String Segment Definition

Scope

Until the next `STRING_SEG` pragma

Syntax

```
#pragma STRING_SEG (<Modif><Name> | DEFAULT)
```

Synonym

`STRING_SECTION`

Arguments

Listing 7.24 Some of the strings which may be used for <Modif>

```
__FAR_SEG      (compatibility alias: FAR)
__PAGED_SEG
```

NOTE Do not use a compatibility alias in new code. It only exists for backwards compatibility. Some of the compatibility alias names conflict with defines found in certain header files. So 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.

<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. 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 7.25](#) is an example of the `STRING_SEG` pragma allocating strings into a segment with the name, `STRING_MEMORY`.

Listing 7.25 Using a `STRING_SEG` pragma to allocate a segment for strings

```
#pragma STRING_SEG __FAR_SEG STRING_MEMORY

char * __far p="String1";

void f(char * __far );

void main(void) {
    f("String2");
}

#pragma STRING_SEG DEFAULT
```

See also

[RS08 Backend](#)

Linker section of the Build Tools manual

[#pragma CONST_SEG: Constant Data Segment Definition](#)

[#pragma DATA_SEG: Data Segment Definition](#)

Compiler Pragmas

Pragma Details

#pragma TEST_CODE: Check Generated Code

Scope

Function Definition

Syntax

```
#pragma TEST_CODE CompareOperator <Size> [<HashCode>]
CompareOperator: ==|!=|<|>|<=|>=
```

Arguments

<Size>: Size of the function to be used in a 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 intentionally 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 7.26](#) and [Listing 7.27](#) present two examples of the `TEST_CODE` pragma.

Listing 7.26 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 7.27](#)).

Listing 7.27 Using a `Test_Code` pragma with the hashcode option

```
/* 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 in the Online Help

#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) because the pragma is valid for the next definition.

See the [RS08 Backend](#) chapter for details.

Example

[Listing 7.28](#) marks the `MyInterrupt()` function as an interrupt function.

Listing 7.28 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 chapter discusses features, restrictions, and further properties of the ANSI-C Compiler Frontend.

- [Implementation Features](#)
- [ANSI-C Standard](#)
- [Floating-Type Formats](#)
- [Volatile Objects and Absolute Variables](#)
- [Bitfields](#)
- [Segmentation](#)
- [Optimizations](#)
- [Using Qualifiers for Pointers](#)
- [Defining C Macros Containing HLI Assembler Code](#)

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 Language – 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)
__paged
```

Far pointers contain a real RS08 address, that is, a linear address between 0 and 0x3FFF. A paged pointer is a 16-bit data entity containing a page number in the high byte and an address in the low byte. The address lies within the paging window, so no additional overhead is required to compute the value. Paged pointers cannot be used to access data across page boundaries. Use paged pointers whenever possible.

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 enable you 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)
```

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 zeros or ones ([Listing 8.6](#)).

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>];
```

where:

- <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 setting the PLACEMENT section in the Linker parameter file (see [Listing 8.8](#)).

Listing 8.8 Assigning global variables to specific addresses

```
#pragma DATA_SEG [__SHORT_SEG] <segment_name>
```

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 an example using correctly and incorrectly the global variable address modifier and [Listing 8.11](#) is a possible PRM file that corresponds with the 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 Linker parameter file settings (prm file)

```
/* the address 0x0500 of "glob" must not be in any address
```

ANSI-C Frontend

Implementation Features

```

    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
#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. Define it in the link parameter file as well (e.g., `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_SEC __SHORT_SEG MY_SHORT_DATA_SEC
#pragma DATA_SEC MY_DATA_SEC
#pragma CONST_SEC MY_CONST_SEC
#pragma DATA_SEC DEFAULT // not necessary,
                          // but good practice
#pragma CONST_SEC DEFAULT // not necessary,
                          // but good practice
int short_var @"MY_SHORT_DATA_SEC"; // OK, accesses are
                                   // short
int ext_var @"MY_DATA_SEC" = 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 an absolute function (e.g., a special function in ROM). [Listing 8.15](#) is a simple example of calling an absolute function using normal ANSI-C.

Listing 8.15 Absolute function

```
#define erase ((void(*) (void)) (0xfc06))
```



ANSI-C Frontend

Implementation Features

```
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, 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; /* ELF: download!    */
static  char L@0x13;    /* no zero out!      */
static  char M@0x14 = 3; /* copy down         */
static const char N@0x15 = 4; /* ELF: download    */
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.

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, [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
```

ANSI-C Frontend

Implementation Features

```
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 MyDirectSet
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, 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. For example:

```
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.

Table 8.1 Interpretation of the `__far` Keyword

Declaration	Allowed	Type Description
<code>int __far f();</code>	OK	<code>__far</code> function returning an <code>int</code>
<code>__far int f();</code>	error	
<code>__far f();</code>	OK	<code>__far</code> function returning an <code>int</code>
<code>int __far *f();</code>	OK	<code>__far</code> function returning a pointer to <code>int</code>
<code>int * __far f();</code>	OK	function returning a <code>__far</code> pointer to <code>int</code>
<code>__far int * f();</code>	error	

ANSI-C Frontend

Implementation Features

Table 8.1 Interpretation of the `__far` Keyword (*continued*)

Declaration	Allowed	Type Description
<code>int __far * __far f();</code>	OK	<code>__far</code> function returning a <code>__far</code> pointer to <code>int</code>
<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 <code>int</code>
<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 <code>int</code>
<code>void * __far (* f)(void)</code>	OK	pointer to function returning a <code>__far</code> pointer to <code>void</code>
<code>void __far * (* f)(void)</code>	OK	pointer to <code>__far</code> function returning a pointer to <code>void</code>

`__near` Keyword

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. Use it in situations where non-qualified pointers are `__far` and you want to specify an explicit `__near` access or when you must explicitly specify the `__near` calling convention.

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 <code>int</code>
<code>int __near __far f();</code>	error	
<code>__near f();</code>	OK	<code>__near</code> function returning an <code>int</code>

Table 8.2 Interpretation of the `__near` Keyword (continued)

Declaration	Allowed	Type Description
<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 ([Listing 8.20](#)):

ANSI-C Frontend

Implementation Features

Listing 8.20 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 8.21](#)).

Listing 8.21 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 8.22](#)).

Listing 8.22 __va_sizeof__ examples

```
__va_sizeof__(char) == sizeof (int)
__va_sizeof__(float) == sizeof (double)
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. These 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 Not all Backends support this keyword. See the [Non-ANSI Keywords](#) section in [RS08 Backend](#).

One of two ways can be used to specify a function as an interrupt routine:

- Use [#pragma TRAP_PROC: Mark function as interrupt function](#) and adapt the Linker parameter file.
- Use the nonstandard `interrupt` keyword.

Use the nonstandard `interrupt` keyword like any other type qualifier ([Listing 8.23](#)). It specifies a function to be an interrupt routine. It is followed by a number specifying the entry in the interrupt vector that contains 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 8.23 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
```

ANSI-C Frontend

Implementation Features

```
// 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 8.24](#)).

Listing 8.24 Examples of the __asm keyword

```
__asm {
    nop
    nop ; comment
}
asm ("nop; nop");
__asm("nop\n nop");
__asm "nop";
__asm nop;
#asm
    nop
    nop
#endasm
```

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:StrictANSI](#) compiler option is not set (see [Listing 8.25](#)).

Listing 8.25 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.

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 depend 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, 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 8.3](#) 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.

ANSI-C Frontend

Implementation Features

Table 8.3 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
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 ³¹	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

Table 8.3 Translation Limitations (ANSI) (continued)

Limitation	Implementation	ANSI-C
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
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

ANSI-C Frontend

Implementation Features

The table below shows other limitations which are not mentioned in an ANSI standard:

Table 8.4 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).
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 value including 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 8.26](#).

Listing 8.26 Operator precedence

Operators	Associativity
<code>() [] -> .</code>	left to right
<code>! ~ ++ -- + - * & (type) sizeof</code>	right to left
<code>& / %</code>	left to right
<code>+ -</code>	left to right
<code><< >></code>	left to right
<code>< <= > >=</code>	left to right
<code>== !=</code>	left to right
<code>&</code>	left to right
<code>^</code>	left to right
<code> </code>	left to right
<code>&&</code>	left to right
<code> </code>	left to right
<code>? :</code>	right to left
<code>= += -= *= /= %= &= ^= = <<= >>=</code>	right to left
<code>,</code>	left to right

Unary `+`, `-` and `*` have higher precedence than the binary forms.

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)
```

Furthermore, is the same as:

`if (0) =>` Therefore, the if condition is always ‘false’.

Hint: 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 8.27 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)
```

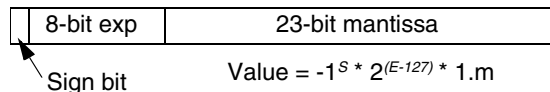
Floating-Type Formats

The RS08 compiler supports only the IEEE32 floating point format. [Figure 8.1](#) shows this format.

Floats are implemented as IEEE32. This may vary for a specific Backend, or possibly, both formats may not be supported.

Figure 8.1 Floating-point formats

IEEE 32-bit format (Precision: 6.5 decimal digits)



Negative exponents are in two's complement; the mantissa is in signed fixed-point format.

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.

ANSI-C Frontend

Floating-Type Formats

For 500, this gives:

```
sign (500.0) = 1,
m, mant (500.0, IEEE) = 1.953125, and
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 to always have positive values:

```
exp (500.0, IEEE32) = 8+127 == 87 (hex) == 10000111 (bin)
```

The fields must be put together as shown [Listing 8.28](#):

Listing 8.28 Representation of decimal 500.0 in IEEE32

```
500.0 (dec) =
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 8.29](#).

Listing 8.29 Representation of decimal -500.0 in IEEE32

```
-500.0 (dec) =
 1 (sign) 10000111 (exponent)
111110100000000000000000 (mantissa) (IEEE32 as bin)
1100 0011 1111 1010 0000 0000 0000 0000 (IEEE32 as bin)
C3 fa 00 00 (IEEE32 as hex)
```

NOTE The IEEE formats recognize several special bit patterns for special values. The number 0 (zero) is encoded by the bit pattern consisting of zero bits only. Other special values such as “Not a number”, “infinity”, -0 (minus zero) and denormalized numbers do exist. Refer to the IEEE standard documentation for details.

Except for the 0 (zero) and -0 (minus zero) special formats, not all special formats may be supported for specific backends.

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 8.30](#) for one reason to use a volatile declaration.

Listing 8.30 Using volatile to avoid an adverse side effect

```
volatile int x;
void main(void) {
  x = 0;
  ...
  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

ANSI-C Frontend

Bitfields

non-portable and inefficient for the masking involved in unpacking individual fields. It is recommended that you use regular bit-and (&) and bit-or (|) operations for I/O port access.

The maximum width of bitfields is Backend-dependent (see [RS08 Backend](#) for details), in that plain `int` bitfields are signed. A bitfield never crosses a word (2 bytes) boundary. As stated in Kernighan and Ritchie's *The C Programming Language*, 2ND 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 of -1 or 0. Consider the following example ([Listing 8.31](#)).

Listing 8.31 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)` 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 8.32](#)).

Listing 8.32 Using unsigned bitfields

```
typedef struct _B {
    unsigned b0: 1;
} B;
    B b;
    if (b.b0 == 1) ...
```

Because `b0` is an unsigned bitfield having the values 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, using bitfields for other purposes is not recommended 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.

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 are allocated into that segment by the Linker. Where that segment actually lies is determined by an entry in the Linker parameter file.

Listing 8.33 Syntax for 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

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>
```

ANSI-C Frontend

Segmentation

```
#pragma STRING_SEG <segment_name>
```

All four pragmas are valid until the next pragma of the same kind is encountered.

In the ELF object file format, constants are always put into a constant segment.

Strings are put into the segment STRINGS until a pragma STRING_SEG 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.

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 `#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.

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
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) */

```

Example of Segmentation with the -Cc Compiler Option

```

static int a;                                /* Placed into Segment: */
static const int c0 = 10;                    /* DEFAULT_RAM(-1) */
                                              /* 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 called optimizations. 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 are 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, ensure that the C source code already contains a shift, or that the value to be shifted is 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 8.34](#)). This results in using less memory space.

Listing 8.34 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 8.35](#)).

Listing 8.35 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 8.5](#) describes how the Compiler implements these strategies.

Table 8.5 Switch Implementations

Method	Description
Branch Sequence	For small switches with scattered case label values, the Compiler generates an <code>if...elsif...elsif...else...</code> sequence if the Compiler switch <code>-Os</code> 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

This section provides some examples for the use of `const` or `volatile` because `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];
```

ANSI-C Frontend

Using Qualifiers for Pointers

`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 an additional one:

```
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.

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:

```
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 uses a pseudo Assembler Language:

```
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 8.36](#)).

Listing 8.36 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 8.37 Invoking the ClearR0 macro.

```
ClearR0;
```

And then the preprocessor expands the macro.

Listing 8.38 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 8.39](#)).

Listing 8.39 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 8.40](#)).

Listing 8.40

```
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 8.41 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 8.42](#)).

Listing 8.42 Calling the ClearR0andR1 macro

```
ClearR0andR1;
```

The preprocessor expands the macro ([Listing 8.43](#)).

Listing 8.43 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 8.44](#) defines the `Clear1` macro that uses the `var` parameter.

Listing 8.44 `Clear1` macro definition.

```
/* This macro initializes the specified variable to 0.*/
#define Clear1(var) {__asm CLR var;}
```

Listing 8.45 Invoking the `Clear1` macro in the source code

```
Clear1(var1);
```

Listing 8.46 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](#), instructs the Compiler that in all the macros defined afterward, the instructions remain unchanged wherever the symbol `#` is specified. This macro is valid for the rest of the file in which it is specified.

Listing 8.47 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;}
```

Listing 8.48 Invoking the Clear2 macro in the source code

```
Clear2(var1);
```

Listing 8.49 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 8.50](#).

Listing 8.50 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; }
```

ANSI-C Frontend

Defining C Macros Containing HLI Assembler Code

Listing 8.51 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 the 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.

Listing 8.52 A pragma anywhere in the header file can enable assembler output

```
#pragma CREATE_ASM_LISTING ON
```

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. Put macros containing elements not supported by the assembler 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.

Example

Listing 8.53 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 8.54](#)):

Listing 8.54 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
```

ANSI-C Frontend

Defining C Macros Containing HLI Assembler Code

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 8.55 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 matters 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 8.56](#)) shows some uses of this feature:

Listing 8.56 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 8.56](#) produces the following output ([Listing 8.57](#)):

Listing 8.57 Assembler listing of [Listing 8.56](#)

```

                                XREF  i
UseI                             EQU   i
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 8.58](#)) do not work or are not generated:

Listing 8.58 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 8.58](#) produces the following output ([Listing 8.59](#)):

Listing 8.59 Assembler listing of [Listing 8.58](#)

```

                                XREF  i
AddressOfI                       EQU   & i
ConstantInt                      EQU   ( ( int ) 1 )
useUndef                         EQU   UndefFkt * 6
Anything                         EQU   § § / % & % / & + * % ç 65467568756 86

```

ANSI-C Frontend

Defining C Macros Containing HLI Assembler Code

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 defined as `Nothing`, with no actual value, 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 object `UndefFkt`. 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.

enums

enums in C have a unique name and a defined value. They are simply generated by the compiler as an `EQU` directive.

Listing 8.60 enum

```
#pragma CREATE_ASM_LISTING ON
enum {
    E1=4,
    E2=47,
    E3=-1*7
};
```

The enum code in [Listing 8.61](#) results in the following EQUs:

Listing 8.61 Resultant EQUs from enums

```
E1          EQU $4
E2          EQU $2F
E3          EQU $FFFFFF9
```

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`. For the name of the size of a `typedef`, an additional term `_SIZE` is appended to the end of the 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 8.62 typedef and struct

```
#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;
```

Creates:

Listing 8.63 Resultant EQUs

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

ANSI-C Frontend

Defining C Macros Containing HLI Assembler Code

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 a 0x1 mask. The offset 2 specifies the most significant bit, which is accessed with a 0x4 mask. 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. Do not generate the function to be called from C, but defined in assembler, 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 8.64](#).

Listing 8.64 Function prototypes

```
#pragma CREATE_ASM_LISTING ON
void main(void);
void f_C(int i, long l);

#pragma CREATE_ASM_LISTING OFF
void f_asm(void);
```

Creates:

Listing 8.65 Functions defined in assembler

```
XREF main
XREF f_C
```

Variables

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 8.66 struct and variable

```
#pragma CREATE_ASM_LISTING ON
struct A {
    char a;
    int i:2;
};
struct A VarA;
#pragma DATA_SEG __SHORT_SEG ShortSeg
int VarInt;
```

Creates:

Listing 8.67 Resultant XREFs and EQUs

	XREF VarA
VarA_a	EQU VarA + \$0
VarA_i	EQU VarA + \$1
VarA_i_BIT_WIDTH	EQU \$2
VarA_i_BIT_OFFSET	EQU \$0
	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 8.68](#) for an example of how C source code with its comments is converted into assembly.

ANSI-C Frontend

Defining C Macros Containing HLI Assembler Code

Listing 8.68 C source code conversion to assembly

```
#pragma CREATE_ASM_LISTING ON
/*
    The main() function is called by the startup code.
    This 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
```

Creates:

```
; 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 files-dependent include files to be created.

- Use `#pragma CREATE_ASM_LISTING ON/OFF` only in C header files. This `#pragma` selects the objects to translate to the assembler include file. The created assembler include file then holds the corresponding assembler directives.
- Do not use the `-La` option as part of the command line options used for all compilations. Use this option in combination with the `-Cx` (no Code Generation) compiler 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.



ANSI-C Frontend

Defining C Macros Containing HLI Assembler Code

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.

-Oi: 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 fun(void);
void main(void) {
    fun();
}
void fun(void) {
    // ...
}

/* better! */
void fun(void) {
    // ...
}
void main(void) {
    fun();
}
```

This also helps the compiler to use a relative branch instruction instead of an absolute branch instruction.

Relocatable Data

The limited RAM size of the RS08 requires careful data allocation. Access global read/write data using tiny, short, direct, or paged addressing. Access global read-only data using paged or far addressing. RS08 non-static local data must be allocated into an OVERLAP section. The compiler does this allocation automatically, using the same address range as is used for direct addressing (0x00 to 0xBF).

- Use tiny addressing (`__TINY_SEG`) for frequently accessed global variables when the operand can be encoded on four bits. The address range for these variables is 0x00 – 0x0F.
- Use short addressing (`__SHORT_SEG`) to access IO registers in the lower RS08 register bank, when the operand can be encoded on five bits. The address range for these variables is 0x00 – 0x1F.
- Use direct (8-bit) addressing (`DEFAULT`) to access global variables when the operand is greater than four or five bits. The address range for these variables is 0x00 to 0xBF.
- Use paged addressing (`__PAGED_SEG`) to access IO registers in the upper RS08 register bank. The address range for these variables is 0x100 to 0x3FF.
- Use paged addressing (`__PAGED_SEG`) to access global read-only (constant) data. Objects allocated in PAGED sections must not cross page boundaries. The address range for these constants is 0x00 to 0x3FFF.
- Use far addressing (`__FAR_SEG`) to access large constant data. Allocate far sections to more than one page. The address range for these constants is 0x00 to 0x3FFF.

See [Listing 9.1](#) for examples using the different addressing modes.

Listing 9.1 Allocating variables on the RS08

```

/* in a header file */
#pragma DATA_SEG __TINY_SEG MyTinySection
char status;

#pragma DATA_SEG __SHORT_SEG MyShortSection
unsigned char IOReg;

#pragma DATA_SEG DEFAULT
char temp;

#pragma DATA_SEG __PAGED_SEG MyShortSection
unsigned char IOReg;
unsigned char *__paged io_ptr = &IOREG;

#pragma DATA_SEG __PAGED_SEG MyPagedSection

```

```
const char table[10];
unsigned char *__paged tblptr = table;

#pragma DATA_SEG __ FAR_SEG MyFarSection
const char table[1000];
unsigned char *__far tblptr = table;
```

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

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.

Using -Ostk

When you use the `-Ostk` option, the compiler analyzes which local variables are alive simultaneously. Based on that analysis the compiler chooses the best memory layout for variables. When using `-Ostk`, two or more variables may end up sharing the same memory location.

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

Generating Compact Code

Programming Guidelines

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 9.3](#)). 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.

Listing 9.3 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 9.4](#)).

Listing 9.4 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.

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;
fun(i++, --i);
```

In the above example, `fun()` is called either with `fun(3,3)` or with `fun(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 must first allocate space for the return value (1) and then to call the function (2). In phase (3) the return value is copied to the variable `s`. In the callee `fun`, during the return sequence, the Compiler must copy the return value (4, `struct copy`).

Depending on the size of the `struct`, this may be done inline. After return, the caller `main` must 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 increase execution time, and increase code size and memory usage.

Listing 9.5 Returning a struct can force the Compiler to produce lengthy code.

```
struct S fun(void)
/* ... */
return s; // (4)
}

void main(void) {
struct S s;
/* ... */
s = fun(); // (1), (2), (3)
/* ... */
}
```

With the example in [Listing 9.6](#), the Compiler just has to pass the destination address and to call `fun` (2). On the callee side, the callee copies the result indirectly into the destination (4). This approach reduces memory 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 9.6 A better way is to pass only a pointer to the callee for the return value.

```
void fun(struct S *sp) {
/* ... */
*sp = s; // (4)
}

void main(void) {
S s;
/* ... */
fun(&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 local memory usage.

Memory access capabilities of the target influences the code quality. Depending on the target capabilities, access efficiency to local variables may vary. Allocating a huge amount of local variables may be inefficient because the Compiler has to generate a complex sequence to allocate the memory in the beginning of the function and to deallocate it in the end ([Listing 9.7](#)):

Listing 9.7 Good candidate for global variables

```
void fun(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 RS08 Compiler overlaps parameter or local variables using a technique called overlapping. The Compiler allocates local variables or parameters as global, and the linker overlaps them depending on their use. Since the RS08 has no stack, this 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 9.8 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 has to handle a count of zero, the `memcpy2()` function in [Listing 9.9](#) is much simpler and faster:

Generating Compact Code

Programming Guidelines

Listing 9.9 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.

Tiny or Short Segments

Whenever possible, place frequently used global variables into a `__TINY_SEG` or `__SHORT_SEG` segment using:

```

#pragma DATA_SEG __SHORT_SEG MySeg
or
#pragma DATA_SEG __TINY_SEG MySeg

```

Qualifiers

Use the `const` qualifier to help the compiler. The `const` objects are placed into ROM for the Freescale object-file format if the `-Cc` compiler option is given.

RS08 Backend

The Backend is the target-dependent part of a Compiler, containing the code generator. This section discusses the technical details of the Backend for the RS08 family.

Non-ANSI Keywords

The following table gives an overview about the supported non-ANSI keywords:

Keyword	Data Pointer	Supported for Function Pointer	Function
<code>__far</code>	no	no	no
<code>__near</code>	no	no	no
<code>interrupt</code>	no	no	yes
<code>__paged</code>	yes	no	no

Data Types

This section describes how the basic types of ANSI-C are implemented by the RS08 Backend.

Scalar Types

All basic types may be changed with the `-T` option. Note that all scalar types (except `char`) have no signed/unsigned qualifier, and are considered signed by default, for example `int` is the same as `signed int`.

The sizes of the simple types are given by the table below together with the possible formats using the `-T` option:

RS08 Backend

Data Types

Type	Default Format	Default Value Range		Formats Available With Option -T
		Min	Max	
char (unsigned)	8bit	0	255	8bit, 16bit, 32bit
singed char	8bit	-128	127	8bit, 16bit, 32bit
unsigned char	8bit	0	255	8bit, 16bit, 32bit
signed short	16bit	-32768	32767	8bit, 16bit, 32bit
unsigned short	16bit	0	65535	8bit, 16bit, 32bit
enum (signed)	16bit	-32768	32767	8bit, 16bit, 32bit
signed int	16bit	-32768	32767	8bit, 16bit, 32bit
unsigned int	16bit	0	65535	8bit, 16bit, 32bit
signed long	32bit	-2147483648	2147483647	8bit, 16bit, 32bit
unsigned long	32bit	0	4294967295	8bit, 16bit, 32bit
signed long long	32bit	-2147483648	2147483647	8bit, 16bit, 32bit
unsigned long long	32bit	0	4294967295	8bit, 16bit, 32bit

NOTE Plain type char is signed. This default is changed with the -T option.

Floating Point Types

The RS08 compiler supports IEEE32 floating point calculations. The compiler uses IEEE32 format for both `float` and `double` types.

The option `-T` may be used to change the default format of float/double.

Type	Default Format	Default Value Range		Formats Available With Option -T
		Min	Max	
float	IEEE32	-1.17549435E-38F	3.402823466E+38F	IEEE32
double	IEEE32	1.17549435E-38F	3.402823466E+38F	IEEE32
long double	IEEE32	1.17549435E-38F	3.402823466E+38F	IEEE32
long long double	IEEE32	1.17549435E-38F	3.402823466E+38F	IEEE32

Pointer Types and Function Pointers

The size of pointer types depends on the memory model selected. The following table gives an overview.

Type	Example	Size
default data pointer	<code>char*</code>	1 byte
default function pointer	<code>void (*)(void)</code>	2 bytes

Structured Types, Alignment

Local variables are allocated in overlapping areas. The most significant part of a simple variable is stored at the low memory address.

Bit Fields

The maximum width of bit fields is 32 bits. The allocation unit is one byte. The Compiler uses words only if a bit field is wider than eight bits. Allocation order is from the least significant bit up to the most significant bit in the order of declaration.

Register Usage

The RS08 Compiler and library use five pseudo registers defined in the table below:

Register	Normal Function
U (or _U)	scratch register
V (or _V)	scratch register
W (or _W)	scratch register
Y (or _Y)	address register of binary operations
Z (or _Z)	scratch register

Parameter Passing

These parameter passing and return value conventions apply for the RS08:

- Registers used for Parameter Passing (only for parameter lists without the ellipsis):
 - if the last parameter is 8-bit large, the parameter is passed in register A
 - all other are parameters passed using the shared section (OVERLAP)
- Parameter naming conventions for the shared OVERLAP section are zero-relative starting from the last parameter. In other words:
 - `__OVL_funcname_p0` indicates the last parameter
 - `__OVL_funcname_p1` indicates the next to the last parameter
 - `__OVL_funcname_p2` indicates the third from the last parameter, and so on.
- Ellipsis (for open parameters): Pass open parameters using a parameter block allocated in the local scope of the caller. The address of this block is passed implicitly together with the last declared parameter. The callee uses this address to get the open parameters:

```
unsigned char __OVL_callerfuncname_pblock
[MAX_BLOCK_SIZE] // MAX_BLOCK_SIZE is the maximum of
bytes needed by a caller to pass open parameters
```

- The following return address is used for non-leaf functions or for functions that call the runtime library):

```
unsigned char __OVL_funcname_ra[2] // 2 bytes for return
address of caller
```

Entry and Exit Code

The RS08 uses an overlapping system, rather than a stack, for memory allocation. [Listing 10.1](#) illustrates correct entry and exit code use.

Listing 10.1 Entry and Exit Code Example

```

10: int f(int x) {
0000 45 SHA

0001 b700 STA __OVL_f_14__PSID_75300003 -----> entry code. Saves the
SPC value. This only happens for non-leaf functions !
0003 42 SLA

0004 b701 STA __OVL_f_14__PSID_75300003:1
11: g(&x);
0006 a600 LDA #__OVL_f_p0
0008 ad00 BSR PART_0_7(g)
12: return x;

000a 4e000f LDX __OVL_f_p1 -----> stores the return value
000d 4e000e MOV __OVL_f_p0,D[X]
0010 2f INCX

0011 4e010e MOV __OVL_f_p0:1,D[X]
0014 b600 LDA __OVL_f_14__PSID_75300003 -----> restore SPC. This only
happens for non-leaf functions !

0016 45 SHA
0017 b601 LDA __OVL_f_14__PSID_75300003:1
0019 42 SLA
13: }

001a be RTS

```

Pragmas

The Compiler provides some pragmas that control the generation of entry and exit code.

TRAP_PROC

The procedure terminates with an RTS instruction instead of an JAL R6. The same effect can be achieved with the interrupt keyword.

NO_ENTRY

Omits generation of procedure entry code.

NO_EXIT

Does not generate procedure exit code.

Interrupt Functions

Interrupt procedures are quite different from other procedures.

- The function returns with a RTS.
- no registers must be saved.
- interrupt functions can either have no arguments or exactly one with either 8 or 16 bit, this argument is passed in R1 (and not in R2 as it would be for other functions).

#pragma TRAP_PROC

Which page registers are saved is determined by `pragma TRAP_PROC`. The syntax of this pragma is

```
#pragma TRAP_PROC
```

Interrupt Vector Table Allocation

The vector table has to be setup with normal C (or assembly) code. The interrupt number feature for the interrupt vector is not supported for the RS08.

Instead an array of vectors has to be allocated and initialized with the address of the handlers and with their initial thread argument.

Segmentation

The Linker memory space may be partitioned into several segments. The Compiler allows attributing a certain segment name to certain global variables or functions which then are 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 basic types of segments, code and data segments, each with a matching pragma:

```
#pragma CODE_SEG <name>
```

```
#pragma DATA_SEG <name>
```

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`:

```
#pragma CODE_SEG DEFAULT
```

```
#pragma DATA_SEG DEFAULT
```

Optimizations

The Compiler applies a variety of code improving techniques commonly called optimizations. 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 `CMPA #0` or using `COMB` instead of `EORB #0xFF`.

Branch Optimizations

The Compiler uses branch instructions with short offsets whenever possible. Additionally, other optimizations for branches are also available.

Constant Folding

Constant folding options only affect constant folding over statements. The constant folding inside of expressions is always done.

Volatile Objects

The Compiler does not do register tracing on volatile objects. Accesses to volatile objects are not eliminated. It also doesn't change word operations to byte operations on volatile objects as it does for other memory accesses.

Programming Hints

The RS08 is an 8/16-bit processor not designed with high-level languages in mind. You must observe certain points to allow the Compiler to generate reasonably efficient code. The following list provides an idea of what is “good” programming from the processor’s point of view.

- Use the `restrict` keyword as a hint for the pointer to thread function argument descriptors.
- Use 8-bit computations unless larger types are absolutely required. The RS08 core is an 8-bit MCU, so 16-bit arithmetic operations are expensive, since they are implemented by runtime calls.
- Limit the number of local variables, since storage space is limited.

Using `unsigned` types instead of `signed` types is better in the following cases:

- Implicit or explicit extensions from `char` to `int` or from `int` to `long`.
- Use types `long`, `float` or `double` only when absolutely necessary. They produce a lot of code.

High-Level Inline Assembler for the Freescale RS08

The HLI (High Level Inline) Assembler provides a means to make full use of the properties of the target processor 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/C++ with the inline assembler. The Compiler does all that work for you. For further information, refer to the RS08 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, shown in various configurations:

```
__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
```

If you use the first form, multiple `__asm` statements are contained 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 and the semicolon starts an assembly comment.

Mixing HLI Assembly and HLL

Mixing High Level Inline (HLI) Assembly with a High Level Language (HLL, for example C or C++) requires special attention. The Compiler does care about used or modified registers in HLI Assembly, thus you do not have save/restore registers which are used in HLI. It is recommended to place complex HLI Assembly code, or HLI Assembly code modifying any registers, into separate functions.

Example:

```
void foo(void) {
    /* some C statements */
    p->v = 1;
    __asm {
        /* some HLI statements destroying registers */
    }
    /* some C statements */
    p->v = 2;
}
```

In the above sequence, the Compiler holds the value of `p` in a register. The compiler will correctly reload `p` if necessary.

Example

A simple example illustrates the use of the HLI-Assembler. Assume the following:

- from points to some memory area
- to points to some other, non-overlapping memory area.

Then we can write a simple string copying function in assembly language as follows:

```
void _CMPS(void) {
    __asm {
        ADD        #128
        STA        _Z
        LDA        _Y
        ADD        #128
        CMP        _Z
    }
}
```

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 a `__asm` directive generated by macros. As

macros always expand to one single line, only the first form of the `__asm` keyword is used in macros:

```
__asm NOP;
```

For example:

```
#define SPACE_OK { __asm NOP; __asm NOP; }
```

Using the second form is invalid:

```
#define NOT_OK { __asm { \  
                    NOP; \  
                    NOP; \  
                } }
```

The macro `NOT_OK` is expanded by the preprocessor to one single line, which is then incorrectly translated because every assembly instruction must be explicitly terminated by a `newline`. Use the pragma `NO_STRING_CONSTR` to build immediates by using `#` inside macros.

Special Features

The following special features are available with the RS08 compiler.

Caller/Callee Saved Registers

The compiler assumes that R1 and R5 remain valid across function calls. Therefore assembly functions must ensure this condition holds when they are called from C code.

Reserved Words

The inline assembler knows a couple of reserved words, which must not collide with user defined identifiers such as variable names. These reserved words are:

- All opcodes (MOV, NOP, ...)
- All register names (A, X, D[X])
- The fixup identifiers:

Name	Address Kind	Description
%HIGH_6_13	Logical Address	Returns the page number corresponding to a given RS08 14-bit address.
%MAP_ADDR_6	Logical Address	Returns the offset within the paging window corresponding to a given RS08 address.
%LOWC	Logical Address	Returns the low byte of an address; checks for overflows
%HIGH	Logical Address	Returns the high byte of an address.
%FIX16	Global Address	16-bit fixup

For these reserved words, the inline assembler is *not* case sensitive, that is JSR is the same as jsr or even JsR. For all other identifiers (labels, variable names and so on) the inline assembler is case sensitive. The following example shows the syntax of the fixup specification:

```
__asm MOV    #%HIGH_6_13(var), __PAGESEL
```

Pseudo-Opcodes

The inline assembler provides some pseudo opcodes to put constant bytes into the instruction stream. These are:

```
DC.B 1      ; Byte constant 1
DC.B 0      ; Byte constant 0
DC.W 12     ; Word constant 12
DC.L 20,23  ; Longword constants
```

Accessing Variables

The inline assembler allows accessing local and global variables declared in C by using their name in the instruction. For global variable names, use the correct fixup specification (usually %LOWC for the low byte and %HIGH for the high byte part).

Constant Expressions

Constant expressions may be used anywhere an *IMMEDIATE* value is expected. The HLI supports the same operators as in ANSI-C code. The syntax of numbers is the same as in ANSI-C.



High-Level Inline Assembler for the Freescale RS08

Special Features

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 */
```

Read the README.TXT located in the library folder for additional information on memory models and library filenames.

NOTE The RS08 and the HC08 share the standard library files. Therefore the RS08 library files are located in:

```
<install>\lib\hc08c
```

Generating 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 the CodeWarrior IDE 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
ctype.c	ctype.h
	errno.h
heap.c	heap.h
	limits.h
math.c, mathf.c	limits.h , ieemath.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 must 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
- 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.

Library Files

Most of the object files of the ANSI library are delivered in the form of an object library.

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.

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.

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.



Library Files
Library Files

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 heap location, heap size, and what happens when heap memory runs out.

All these points can be addressed in the `heap.c` file. The heap is viewed as a large array, and there is a default error handling function. 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 as traps. This means the `signal()` function allows you to set a vector to some function of your own (ideally a `TRAP_PROC`), while the `raise()` function is unimplemented. If you decide to ignore a certain signal, a default handler is installed that does nothing.

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 are unimplemented. If these functions are needed, the programmer must 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()` perform a HALT.

I/O - `printf.c`

The `printf()` library function is unimplemented 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 two disadvantages to 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);
    return i;
}
```

Special Features

*Locales - locale.**

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, switching this macro off is recommended (clear `ENABLE_OVERFLOW_CHECK` to 0).

Library Structure

In this chapter, 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 ([Listing 14.1](#)):

Listing 14.1 Error handling functions

```
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 14.2](#)).

Listing 14.2 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);
```

Library Structure

Memory Block Functions

```
char * strpbrk(const char *s, const char *set);
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 14.3](#)).

Listing 14.3 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 14.4](#)), has been augmented by an alternate interface (and implementation) using type `float`.

Listing 14.4 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 14.5](#)).

Listing 14.5 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);
```

Library Structure

Memory Management

In addition, the ANSI library also defines a couple of functions operating on integral values ([Listing 14.6](#)):

Listing 14.6 ANSI-C Integral functions

```
int    abs(int i);
div_t  div(int a, int b);
long   labs(long l);
ldiv_t ldiv(long a, long b);
```

Furthermore, the ANSI-C library contains a simple pseudo-random number generator ([Listing 14.7](#)) and a function for generating a seed to start the random-number generator:

Listing 14.7 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 ([Listing 14.8](#)):

Listing 14.8 Memory allocation 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 ([Listing 14.9](#)):

Listing 14.9 Generalized searching and sorting functions

```
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 #undefine the macro.

Listing 14.10 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 ([Listing 14.11](#)).

Listing 14.11 Interface for multibyte and wide characters

```
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);
```

Library Structure

System Functions

System Functions

The ANSI standard includes some system functions for raising and responding to signals, non-local jumping, and so on.

Listing 14.12 ANSI-C system functions

```
void      abort(void);
int       atexit(void(* func) (void));
void      exit(int status);
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 functions shown in [Listing 14.13](#), implemented as macros:

Listing 14.13 Macros with variable-length arguments

```
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 14.14](#)).

Listing 14.14 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 *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 14.15](#)).

Listing 14.15 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 14.16](#).

Listing 14.16 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);
```

Library Structure

printf() and scanf()

printf() and scanf()

More conversions are possible for the C functions for reading and writing formatted data. These functions are shown in [Listing 14.17](#).

Listing 14.17 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. Therefore, it is impossible for Freescale to provide an implementation for these features that the user has to specifically implement.

[Listing 14.18](#) contains file I/O functions while [Listing 14.19](#) has functions for the reading and writing of characters. The functions for reading and writing blocks of data are found in [Listing 14.20](#). Functions for formatted I/O on files are found in [Listing 14.21](#), and [Listing 14.22](#) has functions for positioning data within files.

Listing 14.18 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 14.19 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 14.20 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 14.21 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 14.22 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(
```



Library Structure

File I/O

Types and Macros in the Standard Library

This chapter 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 declared 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

float.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
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

Types and Macros in the Standard Library

locale.h

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 {
    char *decimal_point;          /* "C" locale (default) */
                                /* "." */

    /* 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. For example, 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;       /* "" */

    /* 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. */
    har 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 to set. See [Table 15.4](#).

Table 15.4 Constants used with setlocal()

Constant	Description
LC_ALL	Changes the complete locale
LC_COLLATE	Only changes the locale for the strcoll() and strxfrm() functions
LC_MONETARY	Changes the locale for formatting monetary numbers
LC_NUMERIC	Changes the locale for numeric, i.e., non-monetary formatting
LC_TIME	Changes the locale for the strftime() function
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;
```

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 signal() , 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 Type Constants

Constant	Definition
SIGABRT	Abort program abnormally
SIGFPE	Floating point error
SIGILL	Illegal instruction

Table 15.6 Signal Type Constants (continued)

Constant	Definition
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.
<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 fgetpos() and fsetpos() .

[Table 15.9](#) lists the constants defined in `stdio.h`.

Types and Macros in the Standard Library

stdlib.h

Table 15.9 Constants defined in `stdio.h`

Constant	Description
BUFSIZ	Buffer size for setbuf() .
EOF	Negative constant to indicate end-of-file.
FILENAME_MAX	Maximum length of a filename.
FOPEN_MAX	Maximum number of open files.
_IOFBF	To set full buffering in setvbuf() .
_IOLBF	To set line buffering in setvbuf() .
_IONBF	To switch off buffering in setvbuf() .
SEEK_CUR	fseek() positions relative from current position.
SEEK_END	fseek() positions from the end of the file.
SEEK_SET	fseek() positions from the start of the file.
TMP_MAX	Maximum number of unique filenames tmpnam() can generate.

In addition, 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
<code>typedef div_t;</code>	A struct for the return value of div() .
<code>typedef ldiv_t;</code>	A struct for the return value of ldiv() .

[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 rand() .
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 file `string.h` defines only functions and not types or special defines.

The functions are explained below together with all other ANSI functions.

Types and Macros in the Standard Library

assert.h

assert.h

The file `assert.h` defines the [assert\(\)](#) macro. If the `NDEBUG` macro is defined, then `assert` does nothing. Otherwise, `assert` calls the auxiliary function `_assert` if the one 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)
#else
#define assert(EX) ((EX) ? 0 : _assert(__LINE__, __FILE__))
#endif
```

stdarg.h

The file `stdarg.h` defines the type `va_list` and the [va_arg\(\)](#), [va_end\(\)](#), and [va_start\(\)](#) macros. The `va_list` type implements a pointer to one argument of a open parameter list. The `va_start()` macro 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 `va_arg()` macro 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 therefore it must not be released. The `va_start()` and the `va_arg()` macros have a type parameter, which is accessed only with `sizeof()`. So type, but also variables can be used. See [Listing 15.4](#) for an example using `stdarg.h`.

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. Be especially careful when using variables instead of types for `va_arg()`. In the example above, `res= va_arg(list, res)` is not correct unless `res` has 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 `ctype.h`

```
extern unsigned char  _ctype[];
#define  _U  (1<<0)    /* Uppercase      */
#define  _L  (1<<1)    /* Lowercase     */
#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))

```

Types and Macros in the Standard Library

ctype.h

```
#define toupper(c) (islower(c) ? ((c) - 'a' + 'A') : (c))
#define isascii(c) (!(c) & ~127)
#define toascii(c) (c & 127)
#endif /* __OPTIMIZE_FOR_TIME__ */
```

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 global variable <code>errno</code> 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 the following icon appearing in the right margin next to the function's name:

Hardware specific



Functions for file I/O, which also depend on the particular hardware's setup and therefore also are not implemented, are marked by the following icon in the right margin:

File I/O



The Standard Functions

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\(\)](#), and
[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 -32768 , this value is returned and `errno` is set to `ERANGE`.

See also

[fabs\(\) and fabsf\(\)](#)

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 π 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\(\)](#), and
[tan\(\) and tanf\(\)](#)

The Standard Functions

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\(\)](#), and
[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\(\)](#), and
[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\(\)](#) and
[exit\(\)](#)

atan() and atanf()**Syntax**

```
#include <math.h>
double atan (double x);
float  atanf(float x);
```

The Standard Functions

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 $\pi/2$ radian

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 π radian, 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\(\)](#), and

[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\(\)](#) and
[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 the following:

FloatNum	=	Sign{Digit}[.{Digit}][Exp]
Sign	=	[+ -]
Digit	=	<any decimal digit from 0 to 9>
Exp	=	(e E) SignDigit{Digit}

The Standard Functions

Return

`atoi()` returns the converted integer value.

See also

[atof\(\)](#),
[strtod\(\)](#),
[strtol\(\)](#), and
[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 the following:

$$\text{Number} = [+|-]\text{Digit}\{\text{Digit}\}$$

Return

`atoi()` returns the converted integer value.

See also

[atof\(\)](#),
[atol\(\)](#),
[strtod\(\)](#),
[strtol\(\)](#), and
[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 the following:

```
Number = [+|-]Digit{Digit}
```

Return

`atol()` returns the converted `long` value.

See also

[atoi\(\)](#),
[atof\(\)](#),
[strtod\(\)](#),
[strtol\(\)](#), and
[strtoul\(\)](#)

bsearch()

Syntax

```
#include <stdlib.h>

void *bsearch(const void *key,
              const void *array,
              size_t n,
              size_t size,
              cmp_func cmp());
```

The Standard Functions

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 type `cmp_func` can be declared as:

```
typedef int (*cmp_func)(const void *key,
                        const void *data);
```

The comparison function returns an integer according to [Table 16.2](#):

Table 16.2 Return value from the comparison function, `cmp_func()`

Key element value	Return value
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()`. Do not use the default implementation in interrupt routines because it is not reentrant.

Return

`calloc()` returns a pointer to the allocated memory block. If the block cannot be allocated, the return value is `NULL`.

See also

[malloc\(\)](#) and
[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`.

The Standard Functions

See also

[floor\(\) and floorf\(\)](#) and
[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. Express x in radians.

Return

The cosine cos (x)

See also

[acos\(\) and acosf\(\)](#),
[asin\(\) and asinf\(\)](#),
[atan\(\) and atanf\(\)](#),
[atan2\(\) and atan2f\(\)](#),
[sin\(\) and sinf\(\)](#), and
[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.

The Standard Functions

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\(\)](#), and
[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\(\)](#), and
[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\(\)](#) and

[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);
```

The Standard Functions

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\(\)](#), and
[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\(\)](#) and
[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\(\)](#)

The Standard Functions

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 you are 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\(\)](#) and
[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

The Standard Functions

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\(\)](#), and
[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\(\)](#) and
[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\(\)](#) and
[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`.

The Standard Functions

Return

The largest integral number not larger than x .

See also

[ceil\(\) and ceilf\(\)](#) and
[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\(\)](#), and
[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
<code>r</code>	Open the file as a text file for reading.
<code>w</code>	Create a text file and open it for writing.
<code>a</code>	Open the file as a text file for appending
<code>rb</code>	Open the file as a binary file for reading.
<code>wb</code>	Create a file and open as a binary file for writing.
<code>ab</code>	Open the file as a binary file for appending.
<code>r+</code>	Open a text file for updating.
<code>w+</code>	Create a text file and open for updating.
<code>a+</code>	Open a text file for updating. Append all writes to the end.
<code>r+b</code> , or <code>rb+</code>	Open a binary file for updating.
<code>w+b</code> , or <code>wb+</code>	Create a binary file and open for updating.
<code>a+b</code> , or <code>ab+</code>	Open a binary file for updating, appending all writes to the end.

If the mode contains an “r”, but the file does not 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.

The Standard Functions

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\(\)](#) and
[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 occurs, `fprintf()` returns EOF.

See also

[printf\(\)](#) and
[vfprintf\(\)](#), [vprintf\(\)](#), and [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 occurs, `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.

The Standard Functions

Return

The number of items successfully read.

See also

[fgetc\(\)](#),
[fgets\(\)](#), and
[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. Do not use the default implementation in interrupt routines because it is not reentrant.

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 $x = m * 2^{exp}$. 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\(\)](#), and

[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.

The Standard Functions

See also

[fgetc\(\)](#),
[fgets\(\)](#), and
[scanf\(\)](#)

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 [Table 16.5](#).

Table 16.5 Offset position into the file for the `fseek()` function

Mode	Offset position
SEEK_SET	<code>offset</code> bytes from the beginning of the file.
SEEK_CUR	<code>offset</code> bytes from the current position.
SEEK_END	<code>offset</code> 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\(\)](#), and
[fsetpos\(\)](#)

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\(\)](#), and
[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, do not use this value except as an argument to [fseek\(\)](#).

Return

-1, if an error occurred; otherwise the current file position.

The Standard Functions

See also

[fgetpos\(\)](#) and

[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\(\)](#), and

[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.

getc()*File I/O***Syntax**

```
#include <stdio.h>
int getc(void);
```

Description

`getc()` 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);
```

The Standard Functions

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\(\)](#) and
[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\(\)](#) and
[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. [Table 16.6](#) describes the character ranges tested by the functions.

Table 16.6 Appropriate character range for the testing functions

Function	Range Tested
<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 (! - or ~).
<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.

The Standard Functions

Return

TRUE (i.e., 1), if `ch` is in the character class; zero otherwise.

See also

[tolower\(\)](#) and
[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^{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\(\)](#), and

[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);
```

The Standard Functions

Description

`localeconv()` returns 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\(\)](#), and
[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\(\)](#) and
[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

$\log_{10}(x)$, 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\(\)](#) and
[log10\(\) and log10f\(\)](#)

The Standard Functions

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\(\)](#). Do not use the default implementation in interrupt routines because it is not reentrant.

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\(\)](#) and
[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\(\)](#) and
[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.

The Standard Functions

See also

[mblen\(\)](#) and
[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\(\)](#), and
[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\(\)](#), and
[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\(\)](#), and
[strncmp\(\)](#)

The Standard Functions

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\(\)](#) and
[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\(\)](#) and
[memcpy\(\) and 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\(\)](#), and
[time\(\)](#)

modf() and modff()**Syntax**

```
#include <math.h>
double modf(double x, double *i);
float modff(float x, float *i);
```

The Standard Functions

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\(\)](#), and
[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\(\)](#) and
[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\(\)](#), and

[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\(\)](#) and

[vfprintf\(\)](#), [vprintf\(\)](#), and [vsprintf\(\)](#)

The Standard Functions

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, make sure that the expression `f` has no unexpected 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\(\)](#) and

[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 returns an integer according to [Table 16.7](#).

Table 16.7 Return value from the comparison function, `cmp_func()`

Key element value	Return value
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](#).

The Standard Functions

Table 16.8 Possible arguments to the sorting function, `qsort()`

Argument Name	Meaning
array	A pointer to the beginning (i.e., the first element) of the array to be sorted
n	The number of elements in the array
size	The size (in bytes) of one element in the table
<code>cmp()</code>	The comparison function

NOTE Make sure the array contains elements of equal 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`. Do not use the default implementation in interrupt routines because it is not reentrant.

Return

`realloc()` returns a pointer to the new memory block. If the operation cannot be performed, the return value is `NULL`.

The Standard Functions

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\(\)](#) and
[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\(\)](#) and
[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\(\)](#), and
[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.

The Standard Functions

See also

[fgetc\(\)](#),
[fgets\(\)](#), and
[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 (in that case, `buf` points to an array of `BUFSIZ` bytes).

See also

[fflush\(\)](#) and
[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 all or part of the program's locale, depending on `class`. The new locale is given by the character string `loc`. The classes allowed are given in [Table 16.9](#).

Table 16.9 Allowable classes for the `setlocale()` function

Class	Affected portion of program locale
LC_ALL	All classes
LC_COLLATE	strcoll() and strxfrm() functions
LC_MONETARY	Monetary formatting
LC_NUMERIC	Numeric formatting
LC_TIME	strftime() function
LC_TYPE	Character handling and multi-byte character functions

The CodeWarrior IDE 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.

The Standard Functions

See also

[localeconv\(\)](#),
[strcoll\(\)](#),
[strftime\(\)](#), and
[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\(\)](#) and
[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 in [Table 16.11](#).

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 the CodeWarrior IDE, 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.

See also

[raise\(\)](#)

The Standard Functions

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\(\)](#), and
[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\(\)](#), and
[tan\(\) and tanf\(\)](#)

sprintf()

Syntax

```
#include <stdio.h>
int sprintf(char *s, const char *format, ...);
```

Description

`sprintf()` writes formatted output to the `s` string. It evaluates the arguments, converts them according to the specified 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 a 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>|#

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 outputs a blank if the number is positive and a "-" if it is negative. The effect of "#" depends on the Conversion character ([Table 16.12](#)).

The Standard Functions

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.
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 an asterisk "*" 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](#)).

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`

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|%
```

The conversion characters have the following meanings ([Table 16.14](#)):

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.

The Standard Functions

Table 16.14 Meaning of the Conversion Characters (*continued*)

Conversion	Description
e, E	The argument must be a double. It is printed in the form <code>[-]d.ddde±dd</code> (scientific notation). The precision determines the number of fractional digits; the digit to the left of the decimal is <code>!</code> 0 unless the argument is 0.0. The default precision is 6 digits. If the precision is zero and the format specifier <code>"#"</code> 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 <code>[-]ddd.ddd</code> (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>printf</code> chooses either format <code>f</code> or <code>e</code> (or <code>E</code> if <code>G</code> is given), depending on the magnitude of the value. Scientific notation is used only if the exponent is <code>< -4</code> or greater than or equal to the precision.
n	The argument must be a pointer to an <code>int</code> . <code>printf()</code> writes the number of characters written so far to that address. If <code>n</code> is used together with length specifier <code>h</code> or <code>l</code> , the argument must be a pointer to a <code>short int</code> or a <code>long int</code> .
o	The argument, which must be an <code>unsigned int</code> , is printed in octal notation.
p	The argument must be a pointer; its value is printed in hexadecimal notation.
s	The argument must be a <code>char *</code> ; <code>printf()</code> writes the string.
u	The argument, which must be an <code>unsigned int</code> , is written in decimal notation.
x, X	The argument, which must be an <code>unsigned int</code> , is written in hexadecimal notation. <code>x</code> uses lower case letters <code>a</code> to <code>f</code> , while <code>X</code> uses upper case letters.
%	Prints a <code>"%"</code> sign. Give only as <code>"%%"</code> .

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 `sprintf()` size considerably, and therefore the define `LIBDEF_PRINTF_FLOATING` exists. Set `LIBDEF_PRINTF_FLOATING` if no floating point support is used. Some targets contain special libraries without floating point support. The IEEE64 floating point implementation is not supported.

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);
```

The Standard Functions

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 *)
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 <code>d</code> , 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 <code>n</code> is used together with length specifier <code>h</code> or <code>l</code> , 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 <code>d</code> , but the parameter must be a pointer to an unsigned integral type.
x, X	As <code>u</code> , but reads a hexadecimal number.
%	Skips a <code>%</code> sign in the input. Give only as <code>%%</code> .

- 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 ^, 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 ^ 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 uppercase 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   = Sign Digit {Digit}
OctNumber   = Sign 0 {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!

The Standard Functions

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\(\)](#), and
[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\(\)](#), and
[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\(\)](#), and
[strncmp\(\)](#)

strcoll()**Syntax**

```
#include <string.h>
int strcoll(const char *p, const char *q);
```

The Standard Functions

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\(\)](#), and
[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\(\)](#) and
[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\(\)](#), and
[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](#)):

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

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., "2007").
<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\(\)](#), and
[time\(\)](#)

`strlen()`

Syntax

```
#include <string.h>
size_t strlen(const char *s);
```

The Standard Functions

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\(\)](#) and
[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\(\) and memmove\(\)](#) and
[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.

The Standard Functions

See also

[strchr\(\)](#),
[strcspn\(\)](#),
[strrchr\(\)](#), and
[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\(\)](#), and
[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

[streq\(\)](#),
[strespn\(\)](#),
[strpbrk\(\)](#), and
[strchr\(\)](#)

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

[streq\(\)](#),
[strespn\(\)](#),
[strpbrk\(\)](#),
[strchr\(\)](#), and
[strspn\(\)](#)

The Standard Functions

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\(\)](#), and
[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 is not `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\(\)](#), and
[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:

<code>Int_Number</code>	= <code>Dec_Number</code> <code>Oct_Number</code> <code>Hex_Number</code> <code>Other_Num</code>
<code>Dec_Number</code>	= <code>SignDigit{Digit}</code>
<code>Oct_Number</code>	= <code>Sign0{OctDigit}</code>

The Standard Functions

```

Hex_Number      = 0(x|X)Hex_Digit{Hex_Digit}
Other_Num       = 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`.

See also

[atoi\(\)](#),
[atol\(\)](#),
[scanf\(\)](#),
[strtod\(\)](#), and
[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\(\)](#), and
[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.

The Standard Functions

See also

[setlocale\(\)](#),
[strcmp\(\)](#), and
[strcoll\(\)](#)

system()

*Hardware
specific*



Syntax

```
#include <string.h>
int system(const char *cmd);
```

Description

`system()` executes the `cmd` command line

Return

Zero

tan() and tanf()

Syntax

```
#include <math.h>
double tan(double x);
float tanf(float x);
```

Description

`tan()` computes the tangent of `x`. Express `x` 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\(\)](#), and
[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\(\)](#), and
[tan\(\) and tanf\(\)](#)

The Standard Functions

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, the current calendar time is assigned to timer.

Return

The current calendar time.

See also

[clock\(\)](#),
[mktime\(\)](#), and
[strftime\(\)](#)

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\(\)](#) and
[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

[isalnum\(\)](#), [isalpha\(\)](#), [isctrl\(\)](#), [isdigit\(\)](#), [isgraph\(\)](#), [islower\(\)](#), [isprint\(\)](#), [ispunct\(\)](#), [isspace\(\)](#), [isupper\(\)](#), and [isxdigit\(\)](#),
[toupper\(\)](#)

The Standard Functions

toupper()

Syntax

```
#include <ctype.h>
int toupper(int ch);
```

Description

`tolower()` 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

[isalnum\(\)](#), [isalpha\(\)](#), [isctrl\(\)](#), [isdigit\(\)](#), [isgraph\(\)](#), [islower\(\)](#), [isprint\(\)](#), [ispunct\(\)](#), [isspace\(\)](#), [isupper\(\)](#), and [isxdigit\(\)](#),
[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\(\)](#), and
[getchar\(\)](#)

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 into an open parameter list. Calls to `va_arg()` get a parameter of the given type. [Listing 16.1](#) shows how to do it:

Listing 16.1 Calling an open-parameter function

```
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

[wcstombs\(\)](#)

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](#)
- [General Optimization Hints](#)
- [Frequently Asked Questions \(FAQs\), Troubleshooting](#)
- [Frequently Asked Questions \(FAQs\), Troubleshooting](#)
- [EBNF Notation](#)
- [Abbreviations, Lexical Conventions](#)
- [Number Formats](#)
- [Precedence and Associativity of Operators for ANSI-C](#)
- [List of all Escape Sequences](#)

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.

Getting Started

The best way is to create a new project using the New Project Wizard (in the CodeWarrior IDE: Menu *File > New*) or a project from a stationery template. This sets 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.

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](#) may be difficult. Using macros which deal with the cases below ([Listing A.3](#)) is a better way to deal with this.

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
```

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
```

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](#)):

Porting Tips and FAQs

Migration Hints

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 the CodeWarrior IDE, 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 { and } Block Markers`

```

void foo(void) {
    #asm
        nop
    #if 1
    #endasm
    foo();
    #asm
        #endif
        nop
    #endasm
}

```

In such case, the `#asm` and `#endasm` must be ported to `asm { and } block markers` ([Listing A.11](#)).

Porting Tips and FAQs

Migration Hints

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 compiler warnings carefully. 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 desired (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 this 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.

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)
```

Porting Tips and FAQs

Migration Hints

```
#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 terminates 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.

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 75, define the function as in [Listing A.17](#).

Porting Tips and FAQs

Migration Hints

Listing A.17 Definition of the INCcount() interrupt function

```
interrupt 75 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

```

General Optimization Hints

Here are some hints to reduce the size of your application:

- Find out 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 either, you may remove the complete startup code and set up your memory in the 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. Using the [-Li: List of Included Files](#) option to write a log file displays the statistics for each single option.
- Find out if you can use IEEE32 for both 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., 8 bits instead of 16 or 32 bits).
- Look into the map file to check runtime routines, which usually have a ‘_’ prefix. Check for 32-bit integral routines (e.g., `_BMUL`). 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).

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.20](#):

Listing A.20 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.21](#) is FALSE:

Listing A.21 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-compiles 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 ± 2 seconds. If the compiler compiles two files in that time, both will have the same time

Porting Tips and FAQs

Frequently Asked Questions (FAQs), Troubleshooting

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`) or in its system (Win32s) or system32 (Windows 2000®, Windows XP, or Windows Vista™ operating systems) subdirectory. The Win32s distribution also contains `winhlp32.exe`.

To change the association with Windows 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.

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 environment variable [GENPATH: #include “File” Path](#). In addition, you can use the [-I: Include File Path](#) compiler option to specify the include file path. With the CodeWarrior IDE, 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 the CodeWarrior IDE and the ELF/DWARF object-file format (see [-F \(-F2, -F2o\): 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
```

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
```

Porting Tips and FAQs

Frequently Asked Questions (FAQs), Troubleshooting

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 the CodeWarrior IDE, 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.

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.22 EBNF Syntax

```
ProcDecl  = PROCEDURE "(" ArgList ")".
ArgList  = Expression {"," Expression}.
Expression = Term ("*"|" /") Term.
```

Porting Tips and FAQs

EBNF Notation

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. Punctuation symbols of the language described (not of EBNF itself) are quoted (they are terminals, too), while other terminal symbols are printed in **boldface**.

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 3rd 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 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.23

```

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.24](#).

Listing A.24 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.24](#), 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 `F` 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](#)):

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.25](#) has examples of these numerical formats.

Listing A.25 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 */
3.2l /* 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
+ -	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.26 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.26](#), operator-precedence causes the product of ($c * d$) to be added to b , and that sum is then assigned to a .

In [Listing A.27](#), 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.27 Three assignments in one 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 the [DEFAULTDIR: Default Current Directory](#) environment variable).

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 file `mcutools.ini`.

NOTE XXX is a placeholder for the name of the actual backend. For example, for the RS08 compiler, the name of this section would be [RS08_Compiler].

SaveOnExit

Arguments

1/0

Description

Set to 1 to store configuration when the compiler is closed. Clear to 0 otherwise. The compiler does not ask to store a configuration in either case.

SaveAppearance

Arguments

1/0

Description

Set to 1 to store the visible topics when writing a project file. Clear 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 to store the visible topics when writing a project file. Clear to 0 if not. The editor setting contains all information of the Editor Configuration dialog box.

SaveOptions

Arguments

1/0

Description

Set to 1 to save the options when writing a project file. Clear to 0 otherwise. The options also contain the message settings.

RecentProject0, RecentProject1, etc.

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 so different tips show at different calls.

Saved

Always saved when saving a configuration file.

ShowTipOfDay

Arguments

0/1

Description

Show the Tip of the Day dialog box at startup by setting ShowTipOfDay to 1.

1: Show Tip of the Day

0: Show 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 A 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]
```

```
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 for the local editor to use. 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 RS08 compiler, the name of this section would be [RS08_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.

Description

Font attributes.

Local Configuration File Entries

[XXX_Compiler] Section

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`

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.

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.

EditorDDETopicName**Arguments**

Topic name. For example, “system”

Local Configuration File Entries

Example

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_Exec=C:\windows\notepad.exe
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
```


Local Configuration File Entries*Example*

```
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

Known C++ Issues in the RS08 Compilers

Template Issues

This section describes unsupported template features.

- Template specialization is unsupported. Example:

```
template <class T> class C {};
template <> class C<double> {};
-----^----- ERROR
```

- Declaring a template in a class is unsupported. Example:

```
struct S {
    template <class T1, class T2> void f(T1, T2) {}
};

-   template <class T> struct S<...>
-template <int i>
```

- Non-template parameters are unsupported. Example:

```
template<> int f()

-   S03< ::T03[3]> s03;
-----^-----Doesn't know global scope ::

template <int i, class P> struct S {
    S<0xa301, int(*)[4][3]> s0;
-----^-----Wrong type of template argument
```

Known C++ Issues in the RS08 Compilers

Operators

- Implicit instantiations are unsupported. Example:

```

template <int i > struct A{
    A<i>() {}
-----^-----ERROR implicit instantiation
    }
-   void g00(void) {}
        void g00(U) {}
    int  g00(char) { return 0; }
-----^-----ERROR: Function differ in return type

```

- Accepting a template template parameter is unsupported. Example:

```

template <template <class P> class X, class T> struct A{}

```

- Defining a static function template is unsupported. Example:

```

    template <class T> static int f(T t) {return 1}
-----^-----ERROR : Illegal storage class

```

Operators

This section describes operator-related limitations and issues as well as unsupported operator features.

- Relational operators other than ‘==’ are unsupported for function pointers.
- Operators in expressions are unsupported. Example:

```

- struct A { };
  void operator*(A) { counter++; }
  enum B{ };
  int operator*(B) { return 0; }
-----^-----Function differs in return type only
                    (found 'void ' expected 'int ')
- struct A{
    operator int*(){return &global;}
  }
  A a;
  (void)*a;
-----^-----Compile ERROR

```

```
- struct A{};
   struct B:struct A{};
   int operator*(A) {return 1;}
   int f() {
       B b;
       return (*b);
   }
-----^-----Illegal cast operation

- int operator->*(B,int){ return 1; }
-----^-----ERROR: unary operator must have one parameter
```

- When an expression uses an operator, a member function with the operator's name should not hide a non-member function with the same name. Example:

```
struct A {
    void operator*() { }
    void test();
};
void operator*(S, int) { } // not hidden by S::operator*()
void S::test(){
    S s;
    (void) (s * 3);
}
-----^-----Compile ERROR
```

- Explicit operator calls are unsupported. Example:

```
struct B {
    operator int() { return 1; }
};
B b;
b.operator int();
-----^-----ERROR: Not supported explicit operator call
```

Binary Operators

The following binary operator functions are unsupported:

- Implementing the binary `->*` operator as a non-member function with two parameters. Example:

```
friend long operator->* (base x, base y) ;
```

Known C++ Issues in the RS08 Compilers

Operators

- Implementing the binary `->*` operator as a non-static member function with one parameter. Example:

```
int operator ->* (C) ;
```

- Overloaded operators are unsupported. Example:

```
struct S {
    int m;
    template <class T> void operator+=(T t) { m += t; } //
ERROR at template
};
```

Unary operators

The following unary operator functions are unsupported:

- Implementing the unary `~` operator as a non-member function with one parameter. Example:

```
int operator ~(C &X) { return 1; }
int tilda (C &X)      { return 1; }
if (~c != tilda(c))
```

```
-----^-----ERROR: Integer-operand expected
```

- Implementing the unary `!` operator as a non-member function with one parameter. Example:

```
class A{};
int operator!(A &X) { return 1; }
int bang_(A &X) { return 1; }
A a;
if (!(a) != (bang_(a)))
```

```
-----^-----ERROR : Arithmetic type or pointer expected
```

- Logical OR operators are unsupported. Example:

```
class X {
public:
    operator int() {i = 1; return 1;}
} x;
```

```
(void) (0 || x);
-----^-----ERROR
```

- Conditional operators are unsupported. Example:

```
int x = 1;
int a = 2;
int b = 3;
x?a:b = 1;
-----^-----ERROR
```

- Assignment operators are incorrectly implemented. Example:

```
(i = 2) = 3;
-----^----- The result of the = operator shall be an lvalue
(i *= 2) = 3;
-----^----- The result of the *= operator shall be an lvalue
(i += 5) = 3;
-----^----- The result of the += operator shall be an lvalue
```

Equality Operators

The following equality operator features are unsupported.

- Defining a pointer to member function type. Example:

```
struct X {
    void f() {}
};
typedef void (X::*PROC)();
```

- Permitting an implementation to compare a pointer to member operand with a constant expression which evaluates to zero using the == operator.

```
class X {
public:
    int m;
};
(void) ( &X::m == 0 );
-----^-----ERROR
```

Header Files

Header files of type `std namespace` are unsupported.

Included `cname` header files are not mapped to `name.h`. Example:

```
#include <cstring>
-----^----- ERROR
```

[Table D.1](#) shows unimplemented header files.

Table D.1 Unimplemented Header Files

<algorithm>	<iomanip>	<memory>	<streambuf>
<bitset>	<iosfwd>	<new>	<typeinfo>
<climits>	<iostream>	<numeric>	<utility>
<complex>	<istream>	<ostream>	<valarray>
<deque>	<iterator>	<queue>	<vector>
<exception>	<limits>	<sstream>	<wchar.h>
<fstream>	<list>	<stack>	<wctype.h>
<functional>	<map>	<stdexcept>	

Bigraph and Trigraph Support

The compiler does not recognize the trigraph sequence `??!` as equal to `|`.

In some cases the compiler fails to replace the `% :` sequence. Example:

```
#if (4 == 9)
#include <string.h>
%:endif
^----- ERROR (missing endif directive)
```


Known Class Issues

The following section describes known class issues and unimplemented or unsupported features.

- Class Names

Usually, using elaborate type specifiers ensures the validity of both names when you define a class and a function with the same name in the same scope. However, in the RS08 compilers this type of class name definition causes an error. Example:

```
class C { char c; };
void C(int x) { }
int x;
void main()
{
    C(x);
-----^----- ERROR
}
```

- Local classes are unsupported on the RS08 compilers. Example:

```
void f(void)
{
    class C {
        C() { }
    };
}
```

- The class member access feature is unsupported. Example:

```
class X {
public:
    enum E { a, b, c };
} x;
int type(int ) {return INT;}
int type(long ) {return LONG;}
int type(char ) {return CHAR;}
int type(X::E ) {return ENUMX;}

type(x.a);
-----^----- Ambiguous parameters type
```

Known C++ Issues in the RS08 Compilers

Known Class Issues

- Nested class declaration is unsupported, although some accesses and calls may succeed when using nested classes.
- Nested class depths of ten or more are not supported. Example:

```
struct :: A a;
-----^-----ERROR
```

- Function member definitions are not allowed within local class definitions. Example:

```
void f () {
    class A {
        int g();
-----^-----Illegal local function definition
    };
}
```

- Defining a class within a function template is not allowed. Example:

```
template <class T>
struct A {
    void f();
};

template <class T>
void A<T>::f() {
    class B {
        T x;
    };
-----^-----ERROR
}
```

- Unsupported Scope rules for classes

Declaring the name of a class does not ensure that the scope name extends through the declarative regions of classes nested within the first class. Example:

```
struct X4 {
    enum {i = 4};
    struct Y4 {
        int ar[i];
-----^-----ERROR
    }
}
```

- Unimplemented Storage class specifiers

Normally, C++ allows taking the address of an object declared register. Example:

```
register int a;
int* ab = &a;
-----^----- ERROR: Cannot take address of this object
```

- The mutable storage class specifier is unsupported.

Keyword Support

The following keywords are unsupported:

- typeid
- explicit
- typename
- mutable storage class specifier
- Cast keywords:
 - static_cast
 - const_cast
 - reinterpret_cast
 - dynamic_cast

Member Issues

The following member features are either unimplemented, unsupported, or not functioning correctly in the RS08 compilers.

- Pointer to Member
 - Global pointer to member initialization is unimplemented. Example:

```
struct S1{};
struct S2 { int member; };
struct S3 : S1, S2 {};

int S3::*pmi = &S3::member;
-----^----- ERROR
```

Known C++ Issues in the RS08 Compilers

Member Issues

- Accessing or initializing a class member using a `pointer_to_member` from that class is unsupported. Example:

```
class X{
public :
    int a;
};
int main(){
    int X::* p0 = &X::a;
    X obj;
    obj.*p0 = -1;
-----^-----ERROR:Unrecognized member
}
```

- Constructing an array from a pointer to member of a struct is unsupported. Example:

```
int S::* a0[3];
a0[1] = &S::i
-----^-----Failed
```

- Static member – When you refer a static member using the class member access syntax, the object-expression is not evaluated or is evaluated incorrectly. Example:

```
int flag;
struct S {
    static int val(void) { return flag; }
} s;
S* f01() { flag = 101; return &s; }
void main(){
    int g;
    g = f01()->val(); //evaluation failed
}
```

- Non-Static Member Functions
 - Using non-static data members defined directly in their overlying class in non-static member functions is unsupported. Example:

```
class X {
    int var;
public:
    X() : var(1) {}
}
```

```

        int mem_func();
    } x;

int X::mem_func(){
    return var; //returned value should be 1
}

```

- A non-static data member/member function name should refer to the object for which it was called. However, in the RS08 compiler, it does not. Example:

```

class X {
public:
    int m;
    X(int a) : m(a) {}
}
X obj = 2;
int a = obj.m; //should be 2 (but is not)

```

- Member Access Control
 - Accessing a protected member of a base class using a friend function of the derived class is unsupported. Example:

```

class A{
protected:
    int i;
};
class B:public A{
    friend int f(B* p){return p->i};
} ;

```

- Specifying a private nested type as the return type of a member function of the same class or a derived class is unsupported. Example:

```

class A {
protected:
    typedef int nested_type;
    nested_type func_A(void);
};
Class B: public A{
    nested_type func_B(void);
};
A::nested_type A::func_A(void) { return m; }
B:: nested_type B::func_B(void) { return m; }
^-----ERROR: Not allowed

```

Known C++ Issues in the RS08 Compilers

Constructor and Destructor Functions

- Accessing a protected member is unsupported. Example:

```
class B {
protected:
    int i;
};
class C : private B {
    friend void f(void);
};
void f(void) { (void) &C::i;}
-----^-----ERROR: Member cannot be accessed
```

- Access declaration

Base class member access modification is unimplemented in the following case:

```
class A{
public:
    int z;
};

class B: public A{
public:
    A::z;
-----^-----ERROR
};
```

Constructor and Destructor Functions

The compiler does not support the following destructor features:

- When a class has a base class with a virtual destructor, its user-declared destructor is virtual
- When a class has a base class with a virtual destructor, its implicitly-declared destructor is virtual

The compiler does not support the following constructor features:

- Copy constructor is an unsupported feature. Example:

```
class C { int member;};
void f(void) {
    C c1;
    C c2 = c1;
-----^-----ERROR: Illegal initialization of non-aggregate type
}
```

- Using a non-explicit constructor for an implicit conversion (conversion by constructor) is unsupported. Example:

```
class A{
public:
    int m;
    S(int x):m(x){};
};
int f(A a) {return a.m};
int b = f(5) /*value of b should be 5 because of explicit conversion of
f parameter(b = f(A(5)))*/
```

- Directly invoking a virtual member function defined in a derived class using a constructor/destructor of class x is unsupported. Example:

```
class A{
    int m;
    virtual void vf(){};
    A(int) {vf()}
}
class B: public A{
    void vf(){}
    B(int i) : A(i) {}
}
B b(1); // this should result in call to A::vf()
```

Known C++ Issues in the RS08 Compilers

Constructor and Destructor Functions

- Indirectly invoking a virtual member function defined in a derived class using a constructor of class x is unsupported. Example:

```
class A{
    int m;
    virtual void vf(){};
    void gf(){vf();}
    A(int) {gf();}
}
class B: public A{
    void vf(){}
    B(int i) : A(i) {}
}
B b(1); // this should result in call to A::vf()
```

- Invoking a virtual member function defined in a derived class using a ctor-initializer of a constructor of class x is unsupported. Example:

```
class A{
    int m;
    virtual int vf(){return 1;};
    A(int):m(vf()){}
}
class B: public A{
    int vf(){return 2;}
    B(int i) : A(i) {}
}
B b(1); // this should result in call to A::vf()
```

Overload Features

The following overload features are unsupported at this time.

- Overloadable Declarations

Usually, two function declarations of the same name with parameter types that only differ in a parameter that is an enumeration in one declaration, and a different enumeration in the other, can be overloaded. This feature is unsupported at this time.

Example:

```
enum e1 {a, b, c};
enum e2 {d, e};
int g(e1) { return 3; }
int g(e2) { return 4; }
-----^-----ERROR: function redefinition
```

- Address of Overloaded Function

Usually, in the context of a pointer-to-function parameter of a user-defined operator, using a function name without arguments selects the non-member function that matches the target. This feature is unsupported at this time. Example:

```
const int F_char = 100;
int func(char)
{
    return F_char;
}
struct A {} a;
int operator+(A, int (*pfc)(char))
{
    return pfc(0);
}
if (a + func != F_char){}
-----^----- Arithmetic types expected
```

Known C++ Issues in the RS08 Compilers

Overload Features

- Usually, in the context of a pointer-to-member-function return value of a function, using a function name without arguments selects the member function that matches the target. This feature is unsupported at this time. Example:

```

struct X {
    void f (void)  {}
    void f (int)  {}
} x;
typedef void (X::*mfvp) (void);
mfvp f03() {
    return &X::f;
}
-----^-----ERROR:Cannot take address of this object

```

- Usually, when an overloaded name is a function template and template argument deduction succeeds, the resulting template argument list is used to generate an overload resolution candidate that should be a function template specialization. This feature is unsupported at this time. Example:

```

template <class T> int f(T) { return F_char; }
int f(int) { return F_int; }
int (*p00)(char) = f;
-----^-----ERROR: Indirection to
different types ('int (*)(int)' instead of 'int (*)(char)')

```

- Overloading operators is unsupported at this time. Example:

```

struct S {
    int m;
    template <class T> void operator+=(T t) { m += t; } //
ERROR at template
};

```

Conversion Features

The following conversion features are unsupported.

- Implicit conversions using non-explicit constructors are unsupported. Example:

```
class A{
public:
    int m;
    S(int x):m(x) {};
};
int f(A a) {return a.m};
int b = f(5) /*value of b should be 5 because of explicit conversion of
f parameter(b = f(A(5)))*/
```

- Initializations using user-defined conversions are unsupported. Usually, when you invoke a user-defined conversion to convert an assignment-expression of type `cv S` (where `S` is a class type), to a type `cv1 T` (where `T` is a class type), a conversion member function of `S` that converts to `cv1 T` is considered a candidate function by overload resolution. However, this type of situation is unsupported on RS08 compilers. Example:

```
struct T{
    int m;
    T() { m = 0; }
} t;
struct S {
    operator T() { counter++; return t; }
} s00;
T t00 = s00;
-----^-----Constructor call with wrong number of arguments
```

Standard Conversion Sequences

The following standard conversion sequences are unsupported:

- A standard conversion sequence that includes a conversion having a conversion rank. Example:

```
int f0(long double) { return 0; }
int f0(double) { return 1; }
float f = 2.3f;
value = f0(f); //should be 1
-----^----- ERROR ambiguous
```

Known C++ Issues in the RS08 Compilers

Conversion Features

- A standard conversion sequence that includes a promotion, but no conversion, having a conversion rank. Example:

```
int f0(char) { return 0; }
int f0(int) { return 1; }
  short s = 5;
value = f0(s);
-----^----- ERROR ambiguous
```

- A pointer conversion with a Conversion rank. Example:

```
int f0(void *) { return 0; }
int f0(int) { return 1; }
value = f0((short) 0);
-----^----- ERROR ambiguous
```

- User-Defined Conversion Sequences

A conversion sequence that consists of a standard conversion sequence, followed by a conversion constructor and a standard conversion sequence, is considered a user-defined conversion sequence by overload resolution and is unsupported. Example:

```
char k = 'a';
char * kp = &k;
struct S0 {
    S0(...) { flag = 0; }
    S0(void *) { flag = 1; }
};
const S0& s0r = kp;
-----^-----ERROR: Illegal cast-operation
```

Ranking implicit conversion sequences

The following implicit conversion sequence rankings situations are unsupported at this time.

- When s_1 and s_2 are distinct standard conversion sequences and s_1 is a sub-sequence of s_2 , overload resolution prefers s_1 to s_2 . Example:

```
int f0(const char*) { return 0; }
int f0(char*) { return 1; }
value = f0('a');
-----^-----ERROR: Ambiguous
```

- When s_1 and s_2 are distinct standard conversion sequences of the same rank, neither of which is a sub-sequence of the other, and when s_1 converts c^* to b^* (where b is a base of class c), while s_2 converts c^* to a^* (where a is a base of class b), then overload resolution prefers s_1 to s_2 . Example:

```
struct a
struct b : public a
struct c : public b
int f0(a*) { return 0; }
int f0(b*) { return 1; }
c* cp;
value = f0(cp);
-----^-----ERROR:Ambiguous
```

- When s_1 and s_2 are distinct standard conversion sequences neither of which is a sub-sequence of the other, and when s_1 has Promotion rank, and s_2 has Conversion rank, then overload resolution prefers s_1 to s_2 . Example:

```
int f(int) { return 11; }
int f(long) { return 55; }
short aa = 1;
int i = f(aa)
-----^----- ERROR:Ambiguous
```

Explicit Type Conversion

The following syntax use is not allowed when using explicit type conversions on an RS08 compiler:

```
i = int();//A simple-type-name followed by a pair of parentheses
```

The following explicit type conversion features are unsupported at this time:

- Casting reference to a volatile type object into a reference to a non-volatile type object. Example:

```
volatile int x = 1;
volatile int& y = x;
if((int&)y != 1);
-----^-----ERROR
```

Known C++ Issues in the RS08 Compilers

Initialization Features

- Converting an object or a value to a class object even when an appropriate constructor or conversion operator has been declared. Example:

```
class X {
public:
    int i;
    X(int a) { i = a; }
};
X x = 1;
x = 2;
-----^-----ERROR: Illegal cast-operation
```

- Explicitly converting a pointer to an object of a derived class (private) to a pointer to its base class. Example:

```
class A {public: int x;};
class B : private A {
public:
    int y;
};
int main(){
    B b;
    A *ap = (A *) &b;
-----^----- ERROR: BASE_CLASS of class B cannot be accessed
}
```

Initialization Features

The compiler does not support the following initialization features:

- When an array of a class type T is a sub-object of a class object, each array element is initialized by the constructor for T. Example:

```
class A{
public:
    A(){}
};
class B{
public:
    A x[3];
    B(){};
};
B b; /*the constructor of A is not called in order to initialize the
elements of the array*/
```

- Creating and initializing a new object (call constructor) using a new-expression with one of the following forms:
 - (void) new C();
 - (void) new C;
- When initializing bases and members, a constructor's mem-initializer-list may initialize a base class using any name that denotes that base class type (typedef); the name used may differ from the class definition. Example:

```

struct B {
    int im;
    B(int i=0) { im = i; }
};
typedef class B B2;
struct C : public B {
    C(int i) : B2(i) {} ;
-----^-----ERROR
};
  
```

- Specifying explicit initializers for arrays is not supported. Example:

```

typedef M MA[3];
struct S {
    MA a;
    S(int i) : a() {}
-----^-----ERROR: Cannot specify explicit
initializer for arrays
};
  
```

- Initialization of local static class objects with constructor is unimplemented. Example:

```

struct S {
    int a;
    S(int aa) : a(aa) {}
};
static S s(10);
-----^-----ERROR
  
```

See [Conversion Features](#) also.

Known C++ Issues in the RS08 Compilers

Errors

The following functions are incorrectly implemented:

- sprintf
- vprintf
- putc
- atexit from stdlib.h
- strlen from string.h
- IO functions (freopen, fseek, rewind, etc.)

The following errors occur when using C++ with the RS08 compiler.

- EILSEQ is undefined when <errno.h> is included
- Float parameters pass incorrectly


```
int func(float, float, float );
func(f, 6.000300000e0, 5.999700000e0)
the second value becomes -6.0003
```
- Local scope of switch statement is unsupported for the default branch. Example:

```
switch (a){
    case 'a': break;
    default :
        int x = 1;
        -----^-----ERROR: Not declared x
}
```

- An if condition with initialized declaration is unsupported. Example:

```
if(int i = 0)
-----^-----ERROR
```

The following internal errors occur when using C++ with the RS08 compiler:

- Internal Error #103. Example:

```
long double & f(int i ) {return 1;}
long double i;
if (f(i)!=i)
-----^-----Internal Error
```

- Internal Error #385, generated by the following example:

```
class C{
public:
    int n;
    operator int() { return n; };
}cy;
switch(cy) {
-----^-----ERROR
    case 1:
        break;
    default:
        break;
}
```

- Internal Error #418, generated by the following example:

```
#include <time.h>
struct std::tm T;
```

- Internal Error #604, generated by the following example:

```
class C {
    public:
        int a;
        unsigned func() { return 1;}
};

unsigned (C::*pf)() = &C::func;
if (pf != 0 );
-----^-----Generates the error
```

- Internal Error #1209, when using a twelve-dimensional array
- Internal Error #1810, generated by the following example:

```
struct Index {
    int s;
    Index(int size) { s = size; }
    ~Index(void){ ++x; }
};
for (int i = 0; i < 10; i++)
    for (Index j(0); j.s < 10; j.s++) {
        // ...
    }
```

Other Features

This section describes unsupported or unimplemented features.

- Unsupported data types include:
 - `bool`
 - `wchar_t` (wide character).
- Exception handling is unsupported
- Using comma expressions as lvalues is unsupported. Example:


```
(a=7, b) = 10;
```
- Name Features
 - Namespaces are currently unsupported. Example:

```
namespace A {
-----^----- ERROR
  int f(int x);
}
```

- The name lookup feature is currently unsupported. Name lookup is defined as looking up a class as if the name is used in a member function of *X* when the name is used in the definition of a static data member of the class. Example:

```
class C {
public:
    static int i;
    static struct S {
        int i; char c;
    } s;
};
int C::i = s.i;
```

- Hiding a class name or enumeration name using the name of an object, function, or enumerator declared in the same scope is unsupported. Example:

```
enum {one=1, two, hidden_name };
struct hidden_name{int x;};
-----^-----Not allowed
```

- Global initializers with non-const variables are unsupported. Example:

```
int x;
int y = x;
```

- Anonymous unions are unsupported. Example:

```
void f()
{
    union { int x; double y; };
    x = 1;
    y = 1.0;
}
```

- The following time functions (<ctime>) are unsupported:

- time()
- localtime()
- strftime()
- ctime()
- gmtime()
- mktime()
- clock()
- asctime()

- The fundamental type feature is not supported:

```
int fun (char x){}
int fun (unsigned char x){}
```

-----^-----Illegal function redefinition

- Enumeration declaration features

- Defining an enum in a local scope of the same name is unsupported. Example:

```
enum e { gwiz }; // global enum e
void f()
{
    enum e { lwiz };
```

-----^----- ERROR: Illegal enum redeclaration

Known C++ Issues in the RS08 Compilers

Other Features

- The identifiers in an enumerator-list declared as constants, and appearing wherever constants are required, is unsupported. Example:

```
int fun(short l) { return 0; }
int fun(const int l) { return 1; }
enum E { x, y };
fun(x); /*should be 1*/
```

- Unsupported union features:
 - An unnamed union for which an object is declared having member functions
 - Allocation of bit-fields within a class object. Example:

```
enum {two = 2};
struct D { unsigned char : two; };
```

- The following multiple base definition features are unimplemented as yet:
 - More than one indirect base class for a derived class. Example:

```
Class B:public A(){};
Class C: public B(){};
Class D :public B, public A,publicC{};
```

- Multiple virtual base classes. Example:

```
class A{};
class B: public virtual A{};
class C: public virtual A{};
class D: public B, public C{}
```

- Generally, a friend function defined in a class is in the scope of the class in which it is defined. However, this feature is unsupported at this time. Example:

```
class A{
public:
    static int b;
    int f(){return b;};
};
int A::b = 1;
int x = f(); /*ERROR : x!=1 (it should be 1)*/
```

- The compiler considers the following types ambiguous (the same):
 - char
 - unsigned char
 - signed char
- The Call to Named Function feature is unsupported. Example:

```
class A{
    static int f(){return 0;}
    friend void call_f(){
        f();
        -----^-----ERROR: missing prototype (it should be accepted
                    by the compiler)
    }
}
```

- Preprocessing directives are unsupported. Example:

```
#define MACRO (X) 1+ X
MACRO(1) + 1;
-----^-----Illegal cast-operation
```

- The following line control feature is unsupported.
 - Including a character-sequence in a line directive makes the implementation behave as if the content of the character string literal is equal to the name of the source file. Example:

```
#line 19 "testfile.C" //line directive should alter __FILE__
```

- The following floating point characteristics errors occur:
 - Float exponent is inconsistent with minimum
 - `power(FLT_RADIX, FLT_MIN_EXP -1) != FLT_MIN`
 - Float largest radix power is incorrect
 - `FLT_MAX / FLT_RADIX + power(FLT_RADIX, FLT_MAX_EXP - FLT_MANT_DIG -1) != power(FLT_RADIX, FLT_MAX_EXP -1)`
 - Multiplying then dividing by radix is inexact
 - Dividing then multiplying by radix is inexact
 - Double exponent is inconsistent with minimum
 - Double, power of radix is too small

Known C++ Issues in the RS08 Compilers

Other Features

- Double largest radix power is incorrect
- Multiplying then dividing by radix is inexact
- Dividing then multiplying by radix is inexact
- Long double exponent is inconsistent with minimum
- Long double, power of radix is too small
- Long double largest radix power is incorrect
- The following best viable function is unsupported:
 - When two viable functions are indistinguishable implicit conversion sequences, it is normal for the overload resolution to prefer a non-template function over a template function. Example:

```
int f ( short , int ) { return 1; }
template <class T> int f(char, T) { return 2; }
value = f(1, 2);
-----^-----ERROR: Ambiguous
```

- The following Reference features are unsupported:
 - Object created and initialized/destroyed when reference is to a const. Example:

```
const X& r = 4;
-----^-----ERROR: Illegal cast-operation
```

- The following syntax is unsupported:

```
int a7, a;
if(& (::a7) == &a);
-----^-----ERROR:Not supported operator ::
```

- Aggregate features
 - Object initialization fails. Example:

```
class complex{
    float re, im;
    complex(float r, float i = 0) { re=r; im=i; };
    int operator!=( complex x ){
}
complex z = 1;
z!=1
-----^-----ERROR :Type mismatch
```

- Initialization of aggregate with an object of a struct/class publicly derived from the aggregate fails. Example:

```

class A {
    public:
    int a;
    A(int);
};
class B: public A{
    public:
        int b;
        B(int, int);
};
B::B(int c, int d) : A(d) { b = c; }
    B b_obj(1, 2);
int x = B_obj.a;
-----^-----ERROR: x should be 2

```

- Evaluating default arguments at each point of call is an unsupported feature.
- The following typedef specifier is unsupported:

```

typedef int new_type;
typedef int new_type;
-----^-----ERROR: Invalid redeclaration of new_type

```

- This return statement causes an error:

```

return ((void) 1);
-----^-----ERROR

```

- Permitting a function to appear in an integral constant if it appears in a sizeof expression is unsupported. Example:

```

void f() {}
int i[sizeof &f];
-----^-----ERROR

```

Known C++ Issues in the RS08 Compilers

Other Features

- Defining a local scope using a compound statement is an unimplemented feature.

Example:

```
int i = 4;
int main(){
    if ((i != 1) || (:i != 4));
-----^-----ERROR
}
```

- The following Main function is currently unimplemented:

`argv[argc]!=0` (it should be guaranteed that `argv[argc]==0.`)

- The following Object lifetime feature is currently unimplemented:
 - When the lifetime of an object ends and a new object is created at the same location before it is released, a pointer that pointed to the original object can be used to manipulate the new object.
- The following Function call features are unsupported:
 - References to functions feature is not supported. Example:

```
int main(){
    int f(void);
int (&fr)(void) = f;/
}
```

- Return pointer type of a function make ambiguous between `void *` and `X *`. Example:

```
class X {
public:
    X *f() { return this; }
};
int type(void *x) {return VOIDP;}
int type(X *x) {return CXP;}
X x;
type(x.f())
-----^-----ERROR: ambiguous
```

- Incorrect implementation of a member function call when the call is a conditional expression followed by argument list. Example:


```

struct S {
S(){}
    int f() { return 0; }
    int g() { return 11; }
int h() {
    return (this->*((0?(&S::f) : (&S::g))))();
-----^-----ERROR
    }
};

```

- The following Enumeration feature is unsupported:
 - For enumerators and objects of enumeration type, if an `int` can represent all the values of the underlying type, the value is converted to an `int`; otherwise if an `unsigned int` can represent all the values, the value is converted to an `unsigned int`; otherwise if a `long` can represent all the values, the value is converted to a `long`; otherwise it is converted to `unsigned long`. Example:

```

enum E { i=INT_MAX, ui=UINT_MAX , l=LONG_MAX, ul=ULONG_MAX };
-----^-----ERROR: Integral type expected
or enum value out of range

```

- Delete operations have the following restrictions:
 - Use the `S::operator delete` only for single cell deletion and not array deletion. For array deletion, use the global `::delete()`. Example:

```

struct S{
    S() {}
    ~S () {destruct_counter++;}
    void * operator new (size_t size) {
        return new char[size];
    }
    void operator delete (void * p) {
        destruct_counter ++;
        ::delete p;}
};
S * ps = new S[3];
delete [] ps;
-----^-----ERROR: Used delete operator (should use global
::delete)

```

Known C++ Issues in the RS08 Compilers

Other Features

- Global `::delete` uses the class destructor once for each cell of an array of class objects. Example:

```
S * ps1 = new S[5];
::delete [] ps1;
-----^-----ERROR: ~S is not used
```

- Error at declaring delete operator. Example:

```
void operator delete[] (void *p) {};
-----^-----ERROR
```

- The New operator is unimplemented. Example:

```
- void * operator new[] (size_t);
-----^-----ERROR: Operator must be a function
```

- The following Expression fails to initialize the object. Example:

```
int *p = new int(1+(2*4)-3);
-----^-----ERROR: The object is not initialized
```

- Use placement syntax for new int objects. Example:

```
int * p1, *p2;
p1 = new int;
p2 = new (p1) int;
-----^-----ERROR: Too many arguments
```

- The following Multi-dimensional array syntax is not supported:

```
int tab[2][3];
int fun(int (*tab)[3]);
-----^-----ERROR
```

- The following Goto syntax is unsupported:

```
label:
int x = 0;
-----^-----ERROR: x not declared (or typename)
```

- The following Declaration Statement feature is not implemented:
 - Transfer out of a loop, out of a block, or past an initialized `auto` variable involves the destruction of `auto` variables declared at the point transferred from but not at the point transferred to.
- The following Function Syntax features are not supported:
 - Function taking an argument and returning a pointer to a function that takes an integer argument and returns an integer should be accepted. Example:

```
int (*fun1(int))(int a) {}
int fun2(int (*fun1(int))(int)) ()
-----^-----ERROR
```

- Declaring a function `fun` taking a parameter of type integer and returning an integer with typedef is not allowed. Example:

```
typedef int fun(int)
-----^-----ERROR
```

- A `cv-qualifier-seq` can only be part of a declaration or definition of a non-static member function, and of a pointer to a member function. Example:

```
class C {
    const int fun1(short);
    volatile int fun2(long);
    const volatile int fun3(signed);
};

const int (C::*cp1)(short);
-----^----- ERROR:Should be initialized
volatile int (C::*cp2)(long);
-----^----- ERROR: Should be initialized
const volatile int (C::*cp3)(signed);
-----^----- ERROR: Should be initialized
```

- Use of `const` in a definition of a pointer to a member function of a struct should be accepted. Example:

```
struct S {
    const int fun1(void);
    volatile int fun2(void);
    const volatile int fun3(void);
} s;
const int (S::*sp1)(void) = &S::fun1;
if(!sp1);
```

Known C++ Issues in the RS08 Compilers

Other Features

-----^-----ERROR:Expected int

- When using Character literals, the Multi-characters constant is not treated as int.
Example:

```
int f(int i, char c) {return 1;}
f('abcd', 'c');
```

-----^-----ERROR

- The String characteristic “A string is an ‘array of n const char’” is not supported. Example:

```
int type(const char a[]){return 1;}
type("five") != 1 /*Runtime failed*/
```

- Ambiguity Resolution

```
struct S {
    int i;
    S(int b){ i = b;}
};
S x(int(a));
```

-----^-----ERROR: Should have been a function declaration, not an object declaration

- Using const as a qualified reference is an unsupported feature. Example:

```
int i;
typedef int& c;
const c cref = i;// reference to int
```

-----^-----ERROR

Index

Symbols

- ! 142
- # operator 344
- ## operator 344, 385
- \$ 346
- \$() 113
- \${ } 113
- %” modifier 141
- %’ modifier 141

Numerics

- 0b 346

A

- abort 428, 454
- About
 - Assembler status 67
 - Burner status 68
- About button 69, 70, 71
- About dialog box 107
- abs 454
- Absolute
 - Functions 349
 - Paths 65
 - Variables 346
 - and Linking 351
- Absolute files (*.abs) 64
- ABSPATH 94
- acos(), acosf() 455
- AddIncl 143
- @address 346
- Alignment 409
 - __alignof__ 345, 357
- alloc.c 427
- Anonymous unions, unsupported 611
- Ansi 144, 291, 293
- ANSI-C 161, 295
 - Reference Document 343
 - Standard 343
- Application
 - File name, specifying 71

- Programs 61
- Arguments
 - Command line 67, 68
 - Compiler command line 69
 - Linker command line 71
- Array
 - __far 353
- Arrays with unknown size 550
- ars08.exe 61
- asctime 456
- asin, asinf 456
- #asm 360
 - __asm 145, 345, 360
 - _asm 345, 551
- asm 145, 345, 360, 415
- Assembler 415
- Assembler for RS08 preference panel 67
- Assembler Help option 67
- Assembler messages 67
- Assembler options 67
- assert 457
- assert.h 450
- Associativity 566
- atan, atanf 457
- atan2, atan2f 458
- atexit 428, 459
- atof 459
- atoi 460
- atol 461
- auto 344

B

- Batch burner language (*.bbl) files 68
- Batch file 79
 - BfaB 145, 302
 - BfaGapLimitBits 147
 - BfaTSR 148
 - BfaTSROFF 300
 - BfaTSRON 300
- Big Endian 292
 - __BIG_ENDIAN__ 293
- Bigraphs, unsupported 592

-
- bin directory 63
 - Binary Constants 346
 - BIT 309
 - Bit fields 369, 409, 549
 - __BIT_SEG 309
 - Bitfield allocation 298, 549
 - Bitfield type reduction 300
 - __BITFIELD_ defines 146, 149, 298, 299, 300, 302, 549
 - @bool 550
 - Branch
 - Optimization 376
 - Sequence 378
 - Tree 378
 - break 344
 - Browser information 65
 - bsearch 461
 - BUFSIZ 448
 - Build Extras preference panel 65
 - Build tools 61
 - Burner button 68
 - Burner dialog box 68
 - Burner Help button 68
 - Burner messages dialog box 68
 - Burner options
 - ShowBurnerDialog 62
 - Burner options dialog box 68
 - Burner preference panel 68
 - burner.exe 61
 - Buttons
 - About, Compiler 69
 - About, Importer 70
 - About, Linker 71
 - Burner 68
 - Copy Template, Linker 71
 - Help 68
 - Help, Compiler 69
 - Help, Importer 70
 - Help, Linker 71
 - Messages, Burner 68
 - Messages, Compiler 69
 - Messages, Importer 70
 - Messages, Linker 71
 - Options, Burner 68
 - Options, Compiler 69
 - Options, Importer 70
 - Options, Linker 71
 - Smart Slider, Compiler 69
 - Type Sizes, Compiler 69
- C**
- *.c files 131
 - C programming tools
 - ars08.exe 61
 - burner.exe 61
 - crs08.exe 61
 - decoder.exe 61
 - hiwave.exe 61
 - ide.exe 61
 - libmaker.exe 61
 - linker.exe 61
 - maker.exe 61
 - piper.exe 61
 - C++ comments 145, 163
 - C++ Known Issues 587
 - C++ option 150
 - C++ Support option (-C++) 150
 - Caller/Callee saved registers 417
 - calloc 427, 463
 - case 344
 - Casts, unsupported 595
 - Cc 151, 310, 314, 316, 333, 374, 558, 561
 - Ccx 153, 544, 547
 - ceil 463
 - ceilf 463
 - Cf 155
 - char 344, 361
 - CHAR_BIT 443
 - __CHAR_IS_ define groups 250
 - __CHAR_IS_ type information defines 302
 - CHAR_MAX 443
 - CHAR_MIN 443
 - Ci 155, 293
 - Class names, unsupported 593
 - clearerr 464
 - ClientCommand 89
 - clock 464
 - clock_t 449
-

CLOCKS_PER_SEC 449
 -Cni 160, 293
 __CNI__ 161, 293
 -CnMUL 160
 CODE 138
 CODE GENERATION 140
 Code Size 364
 CODE_SECTION 371
 CODE_SEG 371
 CodeWarrior (with COM) 91
 CodeWarrior groups 37
 CodeWarrior IDE 558, 559, 560
 ide.exe 61
 Integration 63
 CodeWarrior project window 36
 CodeWright 88
 Color
 Error messages 260
 Fatal messages 261
 Information messages 262
 User messages 263
 Warning messages 264
 COM 91
 COM files 63
 Comma expressions, unsupported 610
 Command line arguments 67, 68, 69, 70, 71
 comments 551
 Common Source Files 424
 {Compiler} 113
 Compiler
 Configuration 85
 Control 102
 Error feedback 108
 Error messages 106
 Include file 131
 Input File 131
 Input file 107
 Menu 96
 Menu bar 84
 Messages 104
 Option 100
 Option Settings dialog box 100
 Standard Types dialog box 98
 Status bar 84
 Toolbar 83
 Compiler for RS08 preference panel 69
 Compiler help file 69
 Compiler Messages button 69
 Compiler Option Settings dialog box 49
 Compiler Options
 -ShowSmartSliderDialog 62
 Compiler Options dialog box 69
 Compiler Smart Sliders dialog box 69
 Compiler status information 69
 Compiler Type Sizes dialog box 69
 COMPOPTIONS 114, 117, 135
 const 344, 379
 CONST_SECTION 152, 309, 371, 372
 CONST_SEG 309, 371, 372
 Constant Function 400
 continue 344
 Copy Down 425
 copy down 351
 Copy Template button 71
 COPYRIGHT 118
 cos(), cosf() 465
 cosh(), coshf() 465
 Cosmic 543
 -Cp 162
 -Cppe 162
 -Cq 164
 CREATE_ASM_LISTING 312
 crs08.exe 61
 -CswMaxLF 165
 -CswMinLB 167
 -CswMinLF 168
 -CswMinSLB 170, 555
 ctime 466
 CTRL-S (save) 95
 ctype 430
 ctype.h 451
 -Cu 137, 171, 319, 330
 Current directory 112
 Default 119
 CurrentCommandLine 579
 %currentTargetName 66

D

- D 174
- Data allocation defines 292
- Data types, unsupported 610
- DATA_SECTION 313, 371
- DATA_SEG 313, 371
- __DATE__ 291
- decoder.exe 61
- default 344
- Default Directory 569
- default.env 112, 120, 127, 135
- DEFAULTDIR 113, 119, 131
- DefaultDir 569
- #define 344
- #define directive 175, 189
- defined 344
- Defines
 - Data allocation 292
- __DEMO_MODE__ 292
- Dialog boxes
 - RS08 Compiler Option Settings 49
 - Select File to Compile 51
- difftime 466
- DIG 442
- DIRECT 313
- __DIRECT_SEG 313, 338
- Directive
 - #define 189, 344
 - #elif 344
 - #else 344
 - #endif 344
 - #error 344, 346
 - #if 344
 - #ifdef 344
 - #ifndef 344
 - #include 191, 344
 - #line 344
 - #pragma 344
 - Preprocessor 344
 - #undef 344
 - #warning 344, 346
- Directives
 - Preprocessor 344
- Directories

- bin 63
- bin plugins 63
- Display generated command lines in message
 - window 67, 68, 69, 70, 71
- div 467
- div_t 448
- Division 297, 361
- do 344
- DOS 143
- double 344
- __DOUBLE_IS_define groups 251
- __DOUBLE_IS_DSP__ 304
- __DOUBLE_IS_IEEE32__ 303
- download 351

E

- %E modifier 141
- %e modifier 141
- EABI 301
- EBNF 561
- Ec 176
- Editor 577
- Editor_Exe 573, 578
- Editor_Name 573, 577
- Editor_Opts 574, 578
- EditorCommandLine 583
- EditorDDEClientName 583
- EditorDDEServiceName 584
- EditorDDETopicName 583
- EditorType 582
- EDOM 441
- EDOUT 132
- Eencrypt 178
- Ekey 179
- ELF/DWARF 75, 351, 559
- ELF/DWARF object-file format 75
- __ELF_OBJECT_FILE_FORMAT__ 181, 298
- #elif 344
- #else 344
- else 344
- Embedded Application Binary Interface 301
- #endasm 360
- Endian 292
- #endif 344

ENTRIES 351
 enum 344
 __ENUM_IS_define groups 251
 __ENUM_IS_16BIT__ 303
 __ENUM_IS_32BIT__ 303
 __ENUM_IS_64BIT__ 303
 __ENUM_IS_8BIT__ 303
 __ENUM_IS_SIGNED__ 303
 __ENUM_IS_UNSIGNED__ 303
 -Env 180
 %(ENV) modifier 141
 ENVIRONMENT 112, 120
 Environment
 COMPOPTIONS 117, 135
 COPYRIGHT 118
 DEFAULTDIR 113, 119, 131
 ENVIRONMENT 111, 112, 120
 ERRORFILE 121
 File 112
 GENPATH 122, 124, 125, 131, 184
 HICOMPOPTIONS 117
 HIENVIRONMENT 120
 HIPATH 122, 125
 INCLUDETIME 123
 LIBPATH 122, 124, 128, 131, 132, 184
 LIBRARYPATH 124, 131, 132, 184
 OBJPATH 125, 132
 TEXTPATH 126, 185, 196, 202
 TMP 127
 USELIBPATH 128
 USERNAME 129
 Variable 117
 Environment Variable 307
 Environment Variable section 111
 Environment Variables section 94, 112
 EOF 448
 EPROM 351
 EPSILON 442
 ERANGE 441
 errno 441
 errno.h 441
 Error
 Handling 431
 Listing 132
 Messages 106
 #error 344, 346
 Error Format
 Verbose 267
 ERRORFILE 121
 Escape Sequences 568
 Exception handling, unsupported 610
 exit 428, 467
 EXIT_FAILURE 449
 EXIT_SUCCESS 449
 exp 468
 expf 468
 Explorer 78
 Explorer, launch tool using 112
 Extended Backus-Naur Form, see EBNF
 extern 344

F
 %f modifier 141
 -F1 298, 373
 -F2 181, 298, 373
 F2 shortcut 83
 -F2o 181
 fabs 221, 469
 fabsf 221, 469
 FAR 309, 313, 338, 413
 @far 550
 __far 345, 351, 407
 Arrays 353
 Keyword 352
 far 345, 351
 __FAR_SEG 309, 313, 338
 fclose 469
 feof 470
 ferror 470
 fflush 471
 fgetc 471
 fgetpos 472
 fgets 473
 -Fh 181
 FILE 447
 File
 Environment 112
 Include 131

Object 132
Source 131
File Manager, launch tool using 112
File Names 364
__FILE__ 291
FILENAME_MAX 448
Files
 Absolute (*.abs) 64
 Batch
 regservers.bat 63
 Batch burner language (*.bbl) 68
 COM 63
 *.lib 64
 Library 64
 Object 132
 PRM 45
float 344
float.h 441
__FLOAT_IS_ define groups 251
__FLOAT_IS_DSP__ 303
__FLOAT_IS_IEEE32__ 303
Floating Point 409
floor 473
floorf 473
FLT_RADIX 441
FLT_ROUNDS 441
fmod 474
fopen 474
FOPEN_MAX 448
for 344
fpos_t 447
fprintf 476
fputc 476
fputs 477
fread 477
free 427, 478
freopen 478
frexp 479
frexpf 479
Front End 343
fscanf 479
fseek 480
fsetpos 481
ftell 481

Function Pointer 409
fwrite 482

G

Generating

 Browser information 65
 Disassembly listing 67
GENPATH 94, 122, 124, 125, 131, 184, 558
getc 482
getchar 483
getenv 483
gets 483
Global initialization file 86
Global initializers, unsupported 611
gmtime 484
goto 344, 364
Groups, CodeWarrior 37

H

-H 182, 559, 561
*.h files 131
HALT 427, 428
Header files, unmapped 592
heap.c 427
Help button 68, 69, 70, 71
Help, Assembler 67
Hexadecimal Constants 346
HICOMPONENTS 117
HIENVIRONMENT 120
HIPATH 122
__HIWARE__ 292
hiwave.exe 61, 65
HOST 138, 140
How to Generate Library 423
HUGE_VAL 446

I

-I 558
-I option 131, 183
I/O Registers 351
Icon 78
ide.exe 61
IEEE 409

#if 344
 if 344
 #ifdef 344
 #ifndef 344
 Implementation Restriction 361
 Importer command line arguments 70
 Importer for RS08 preference panel 70
 Importer help file 70
 Importer Messages dialog box 70
 Importer Options dialog box 70
 Importer status information 70
 #include 191, 344
 Include Files 131, 364
 INCLUDETIME 123
 *.ini files 85
 INLINE 220, 315
 inline 397
 Inline Assembler, see Assembler
 inline expansion, enabling 220
 INPUT 138, 140
 int 344
 __INT_IS_ define groups 250
 __INT_IS_16BIT__ 303
 __INT_IS_32BIT__ 303
 __INT_IS_64BIT__ 303
 __INT_IS_8BIT__ 303
 INT_MAX 443
 INT_MIN 443
 Internal IDs 364
 __Interrupt 345
 Interrupt 359, 360, 551
 keyword 359
 vector 359
 @interrupt 551
 __interrupt 359
 interrupt 345, 407
 Interrupt Procedure 411
 INTO_ROM 152, 316
 _IIOFBF 448
 _IIOBFB 448
 _IIONBF 448
 IPATH 125
 isalnum 485
 isalpha 485
 iscntrl 485
 isdigit 485
 isgraph 485
 islower 485
 isprint 485
 ispunct 485
 isspace 485
 isupper 485
 isxdigit 485

J

jmp_buf 446
 Jump Table 378

L

-La 184
 Labels 364
 labs 486
 LANGUAGE 140
 -Lasm 186
 -Lasmc 187
 Lazy Instruction Selection 413
 *.lcf file 548
 lconv 444
 ldexp 486
 ldexpf 486
 -Ldf 188, 291
 ldiv 487
 ldiv_t 448
 Lexical Tokens 364
 -Li 190
 .lib file 64
 *.lib files 64
 libmaker, selecting 64
 libmaker.exe 61
 LIBPATH 94, 122, 124, 128, 131, 132, 184
 Library files 64, 423, 425
 LIBRARYPATH 124, 131, 132, 184
 -Lic 191
 -LicA 192
 -LicBorrow 193
 -LicWait 195
 Limits
 Translation 361

limits.h 443
 #line 344
 Line continuation 116
 __LINE__ 291
 LINK_INFO 318
 Linker files 45
 Linker for RS08 preference panel 44, 71
 Linker help file 71
 Linker Messages dialog box 71
 Linker Options dialog box 71
 Linker status information 71
 linker.exe 61
 Little Endian 292
 __LITTLE_ENDIAN__ 293
 -Ll 196
 -Lm 197
 -LmCfg 199
 -Lo 201
 Local classes, unsupported 593
 locale.h 444
 localeconv 487
 Locales 430
 localtime 488
 log 488
 log10 489
 log10f 489
 logf 488
 long 344
 __LONG_DOUBLE define groups 251
 __LONG_DOUBLE type information
 defines 304
 __LONG_IS define groups 251
 __LONG_IS type information defines 303
 __LONG_LONG define groups 251
 __LONG_LONG type information defines 303
 __LONG_LONG_DOUBLE define groups 251
 __LONG_LONG_DOUBLE type information
 defines 304
 LONG_MAX 443
 LONG_MIN 443
 longjmp 490
 LOOP_UNROLL 319
 -Lp 202
 -LpCfg 203

-LpX 205
 *.lst file 425
 lvalues unsupported 610

M

Macro

 Definition 175
 Expansion 364
 Predefined 291
 maker.exe 61
 malloc 427, 490
 MANT_DIG 442
 mark 320
 math.h 446, 513
 MAX 442
 MAX_10_EXP 442
 MAX_EXP 442
 MB_LEN_MAX 443, 449
 mblen 428, 491
 mbstowcs 428, 491
 mbtowc 428, 492
 *.mcp files 560
 mcutools.ini 86, 114
 memchr 492
 memcmp 493
 memcpy 222, 494
 memmove 494
 memset 222, 494
 MESSAGE 140, 322
 Message format
 Microsoft 267
 MESSAGES 138
 Messages button 67, 68, 69, 70, 71
 Messages, Assembler 67
 Microsoft Developer Studio 89
 Microsoft message format 267
 Microsoft Visual Studio, integrating 73
 MIN 442
 MIN_10_EXP 442
 MIN_EXP 442
 Missing Prototype 551
 mktime 495
 modf 495
 modff 495

__MODULO_IS_POSITIV__ 298
 Modulus 297, 361
 msdev 89
 MS-DOS 143
 mutable unsupported 595
 __MWERKS__ 292

N

-N 206
 %N modifier 141
 %n modifier 141
 Name lookup, unsupported 610
 NAMES 558
 Namespaces, unsupported 610
 NEAR 413
 __near 345, 356, 407
 near 345, 356
 NO_ENTRY 323, 412, 416
 NO_EXIT 325, 412
 NO_FRAME 327
 NO_INLINE 328
 NO_LOOP_UNROLL 329
 __NO_RECURSION__ 305
 NO_RETURN 330
 NO_STRING_CONSTR 332, 384
 -NoBeep 207
 -NoDebugInfo 208
 -NoPath 209
 NULL 447
 Numbers 364

O

*.o files 132
 -Oa 210, 212
 Object
 File 132
 Object files 132
 Object-file formats 75
 -ObjN 212
 OBJPATH 94, 125, 132
 -Od 215
 -Odb 216
 -OdocF 137, 139, 218, 555
 -Odocf 294
 offsetof 447
 -Oi 137, 220
 -Oilib 221
 -OnB 223
 -OnBRA 224
 ONCE 333
 -OnCopyDown 230
 -OnCstVar 231
 -Onp 232
 -OnPMNC 233
 -Onr 234
 -Ont 235
 Operator
 # 344
 ## 344
 operator
 defined 344
 OPTIMIZATION 138, 140
 Optimization
 Branches 376
 Lazy Instruction Selection 413
 Shift optimizations 376
 Strength Reduction 376
 Time vs. Size 211
 Tree Rewriting 377
 __OPTIMIZE_FOR_SIZE__ 211, 293
 __OPTIMIZE_FOR_TIME__ 211, 293
 OPTION 294, 334
 Option
 CODE 138
 CODE GENERATION 140
 HOST 138, 140
 INPUT 138, 140
 LANGUAGE 140
 LANGUAGE 138
 MESSAGE 140
 MESSAGES 138
 OPTIMIZATION 138, 140
 OUTPUT 138, 140
 STARTUP 138
 TARGET 138
 VARIOUS 138, 140
 Option scope 139
 __OPTION_ACTIVE__ 294

-
- Options 569, 582
 - C++ Support (-C++) 150
 - Startup 62, 621
 - Startup command line 61
 - Options button 67, 68, 69, 70, 71
 - Options, Assembler 67
 - Os 211, 293, 378
 - Ot 211, 293
 - OUTPUT 138, 140

 - P**
 - %p modifier 141
 - Panels
 - Assembler for RS08 preference 67
 - Build Extras preference 65
 - Burner preference 68
 - Compiler for RS08 preference 69
 - Importer for RS08 preference 70
 - Linker for RS08 preference 44, 71
 - Target Settings 42
 - Target Settings preference 64
 - Parsing Recursion 364
 - Path list 115
 - Paths
 - Absolute 65
 - Relative 65
 - Pe 244
 - perror 496
 - Pio 246
 - piper.exe 61
 - PLACEMENT 545
 - __PLAIN_BITFIELD define groups 251, 301, 302, 304
 - plugins directory 63
 - Pointer
 - Compatibility 357
 - __far 352
 - Pointer Type 409
 - Pointers, unsupported 595
 - pow 496
 - powf 496
 - #pragma 344
 - CODE_SECTION 371
 - CODE_SEG 371, 413
 - CONST_SECTION 152, 371, 372
 - CONST_SEG 309, 371, 372, 545
 - CREATE_ASM_LISTING 312
 - DATA_SECTION 371
 - DATA_SEG 313, 371, 545
 - FAR 413
 - INLINE 220, 315
 - INTO_ROM 152, 316
 - LINK_INFO 318
 - LOOP_UNROLL 319
 - mark 320
 - MESSAGE 322
 - NEAR 413
 - NO_ENTRY 323, 412, 416
 - NO_EXIT 325, 412
 - NO_FRAME 327
 - NO_INLINE 328
 - NO_LOOP_UNROLL 329
 - NO_RETURN 330
 - NO_STRING_CONSTR 332, 384
 - ONCE 333
 - OPTION 294, 334
 - REALLOC_OBJ 336
 - SHORT 413
 - STRING_SEG 338
 - TEST_CODE 340
 - TRAP_PROC 342, 411, 412
 - SAVE_ALL_REGS 412
 - SAVE_NO_REGS 412
 - #pragma section 545
 - Precedence 566
 - Predefined macros 291, 298
 - Preprocessor directives 344
 - printf 428, 497
 - printf.c 428
 - PRM file 45
 - Procedure
 - Interrupt 411
 - Variable, see Function Pointer
 - Prod 115, 247
 - Prod option 115
 - __PRODUCT_HICROSS_PLUS__ 292
 - {Project} 113
 - Project files, analyzing 37
-

project.ini 115, 118, 135
 %projectFileDir 66
 %projectFileName 66
 %projectFilePath 66
 %projectSelectedFiles 66
 __PTR_SIZE_1__ 305
 __PTR_SIZE_2__ 305
 __PTR_SIZE_3__ 305
 __PTR_SIZE_4__ 305
 ptrdiff_t 295, 447
 __PTRDIFF_T_IS_CHAR__ 296, 297
 __PTRDIFF_T_IS_INT__ 296, 297
 __PTRDIFF_T_IS_LONG__ 296, 297
 __PTRDIFF_T_IS_SHORT__ 296, 297
 __PTRMBR_OFFSET_ define groups 251
 putc 498
 putchar 498
 puts 498
 PVCS 128

Q

qsort 499
 -Qvpt 249

R

raise 500
 rand 501
 RAND_MAX 449
 realloc 427, 501
 REALLOC_OBJ 336
 RecentCommandLine 579
 Recursive comments 551
 register 344
 regservers.bat 63
 Relational operators, unsupported 588
 Relative paths 65
 remove 502
 rename 502
 Restriction
 Implementation 361
 return 344
 rewind 503
 RGB 261, 262, 263, 264
 ROM 379, 558

ROM libraries 425
 ROM_VAR 152, 374
 RS08 compiler macros 302
 RS08 Simulator 60
 RS08 Simulator Startup 60
 __RS08__ 305

S

SAVE_ALL_REGS 412
 SAVE_NO_REGS 412
 SaveAppearance 570
 SaveEditor 570
 SaveOnExit 570
 SaveOptions 571
 scanf 503
 SCHAR_MAX 443
 SCHAR_MIN 443
 SEEK_CUR 448
 SEEK_END 448
 SEEK_SET 448
 Segment 412
 Segmentation 371
 @ “SegmentName” 348
 Select File to Compile dialog box 51
 Service Name 89
 setbuf 504
 setjmp 504
 setjmp.h 446
 setlocale 505
 setvbuf 506
 Shift optimizations 376
 SHORT 313, 413
 short 344
 __SHORT_IS define groups 250
 __SHORT_IS_ define groups 250
 __SHORT_IS_16BIT__ 303
 __SHORT_IS_32BIT__ 303
 __SHORT_IS_64BIT__ 303
 __SHORT_IS_8BIT__ 303
 __SHORT_SEG 313, 372
 -ShowAboutDialog 62
 -ShowBurnerDialog 62
 ShowConfigurationDialog 62
 -ShowMessageDialog 62

- ShowOptionDialog 62
- ShowSmartSliderDialog 62
- ShowTipOfDay 572
- SHRT_MAX 443
- SHRT_MIN 443
- sig_atomic_t 446
- SIG_DFL 446
- SIG_ERR 446
- SIG_IGN 446
- SIGABRT 446
- SIGFPE 446
- SIGILL 446
- SIGINT 447
- signal 507
- signal.c 427
- signal.h 446
- Signals 427
- signed 344
- SIGSEGV 447
- SIGTERM 447
- sin 508
- sinf 508
- sinh 508
- Size
 - Type 407
- size_t 295, 447
- __SIZE_T_IS_UCHAR__ 296, 297
- __SIZE_T_IS_UINT__ 296, 297
- __SIZE_T_IS_ULONG__ 296, 297
- __SIZE_T_IS_USHORT__ 296, 297
- sizeof 344
- Smart Control dialog box 102
- Smart Sliders button 69
- Source File 131
- Source files 131
- %sourceFileDir 66
- %sourceFileName 66
- %sourceFilePath 66
- %sourceLineNumber 66
- %sourceSelection 66
- %sourceSelUpdate 66
- Special Modifiers 141
- sprintf 509
- sqrt 513
- srand 513
- sscanf 514
- Standard Types 98
- Standard types 98
- Start option 79
- STARTUP 138
- Startup 115
 - RS08 Simulator 60
- Startup command line options 61
- Startup Files 425
- Startup options 62, 621
- startup.c 425
- static 344
- StatusbarEnabled 580
- stdarg 357
- stdarg.h 357, 450
- __STDC__ 145, 291, 293
- stddef.h 447
- stderr 448
- stdin 448
- stdio.h 447
- stdlib. 428
- stdlib.c 428
- stdlib.h 448, 513
- stdout 288, 448
- Storage class specifiers, unsupported 595
- strcat 518
- strchr 518
- strcmp 519
- strcoll 519
- strcpy 520
- strcspn 520
- Strength Reduction 376
- strerror 521
- strftime 522
- String concatenation 385
- string.h 449
- STRING_SECTION 338
- STRING_SEG 338
- Strings 351
- strlen 221, 523
- strncat 524
- strncmp 524

strncpy 525
 strpbrk 525
 strrchr 526
 strspn 526
 strstr 527
 strtod 527
 strtok 528
 strtol 529
 strtoul 530
 struct 344
 strxfrm 531
 switch 344
 %symFileDir 66
 %symFileName 66
 %symFilePath 66
 Synchronization 78
 {System} 113
 system 532

T

-T 250, 407, 409
 tan 532
 tanf 532
 tanh 533
 tanhf 533
 TARGET 138
 Target Settings preference panel 42, 64
 %targetFileDir 66
 %targetFileName 66
 %targetFilePath 66
 Template Specialization, unsupported 587
 Termination 78
 TEST_CODE 340
 TEXTPATH 94, 126, 185, 196, 202, 203
 time 534
 time.h 449
 __TIME__ 291
 time_t 449
 @tiny 550
 Tip of the Day 79
 TipFilePos 572
 TipTimeStamp 572
 TMP 127
 TMP_MAX 448

tmpfile 534
 tmpnam 535
 tolower 535
 Tool locations 61
 ToolbarEnabled 580
 Topic Name 89
 toupper 536
 Translation Limits 361
 TRAP_PROC 342, 411, 412, 551
 __TRIGRAPHS__ 156, 293
 Type
 Alignment 409
 Bit fields 409
 Floating Point 409
 Pointer 409
 Size 407
 Size reduction 300
 Type Declarations 364
 Type Sizes button 69
 typedef 344

U

UCHAR_MAX 443
 UINT_MAX 443
 ULONG_MAX 443
 UltraEdit 89
 #undef 344
 ungetc 536
 union 344
 UNIX 112
 unsigned 344
 Use custom PRM file (Linker) 71
 Use Decoder to generate Disassembly Listing 67, 69
 Use External Debugger check box 65
 Use template PRM file (Linker) 71
 USELIBPATH 128
 USERNAME 129
 USHRT_MAX 443

V

-V 256
 va_arg 357, 537
 va_end 537

__va_sizeof__ 345, 358
 va_start 537
 VARIOUS 138, 140
 VECTOR 359
 __VERSION__ 292
 vfprintf 538
 -View 257
 Visual C++ 73
 void 344
 volatile 344, 369
 vprintf 538
 vsprintf 428, 538
 __VTAB_DELTA__ define groups 251
 __VTAB_DELTA__ type information defines 304

W

-W1 288
 -W2 289, 548
 /wait option 79
 #warning 344, 346
 wchar_t 295, 447
 __WCHAR_T_IS_UCHAR__ 296
 __WCHAR_T_IS_UINT__ 296
 __WCHAR_T_IS_ULONG__ 296
 __WCHAR_T_IS_USHORT__ 296
 wcstombs 428, 539
 wctomb 428, 538
 -WErrFile 258
 while 344
 WindowFont 581
 WindowPos 581
 Windows 112
 CodeWarrior project 36
 Winedit 88
 -Wmsg8x3 259
 -WmsgCE 260
 -WmsgCF 261
 -WmsgCI 262
 -WmsgCU 263
 -WmsgCW 264
 -WmsgFb 260, 264, 268, 270, 273, 274, 276
 -WmsgFbi 264
 -WmsgFbm 264
 -WmsgFi 260, 266, 273, 274, 276
 -WmsgFim 266
 -WmsgFiv 266
 -WmsgFob 268, 273
 -WmsgFoi 270, 274, 276
 -WmsgFonf 273
 -WmsgFonp 270, 273, 274, 275, 276
 -WmsgNe 276
 -WmsgNi 277
 -WmsgNu 278
 -WmsgNw 280
 -WmsgSd 281
 -WmsgSe 282
 -WmsgSi 283
 -WmsgSw 284
 -WOutFile 285
 -Wpd 286
 -WStdout 287

Z

Zero Out 425
 zero out 351

